

P&S Heterogeneous Systems

Parallel Patterns: Graph Search

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

19 December 2022

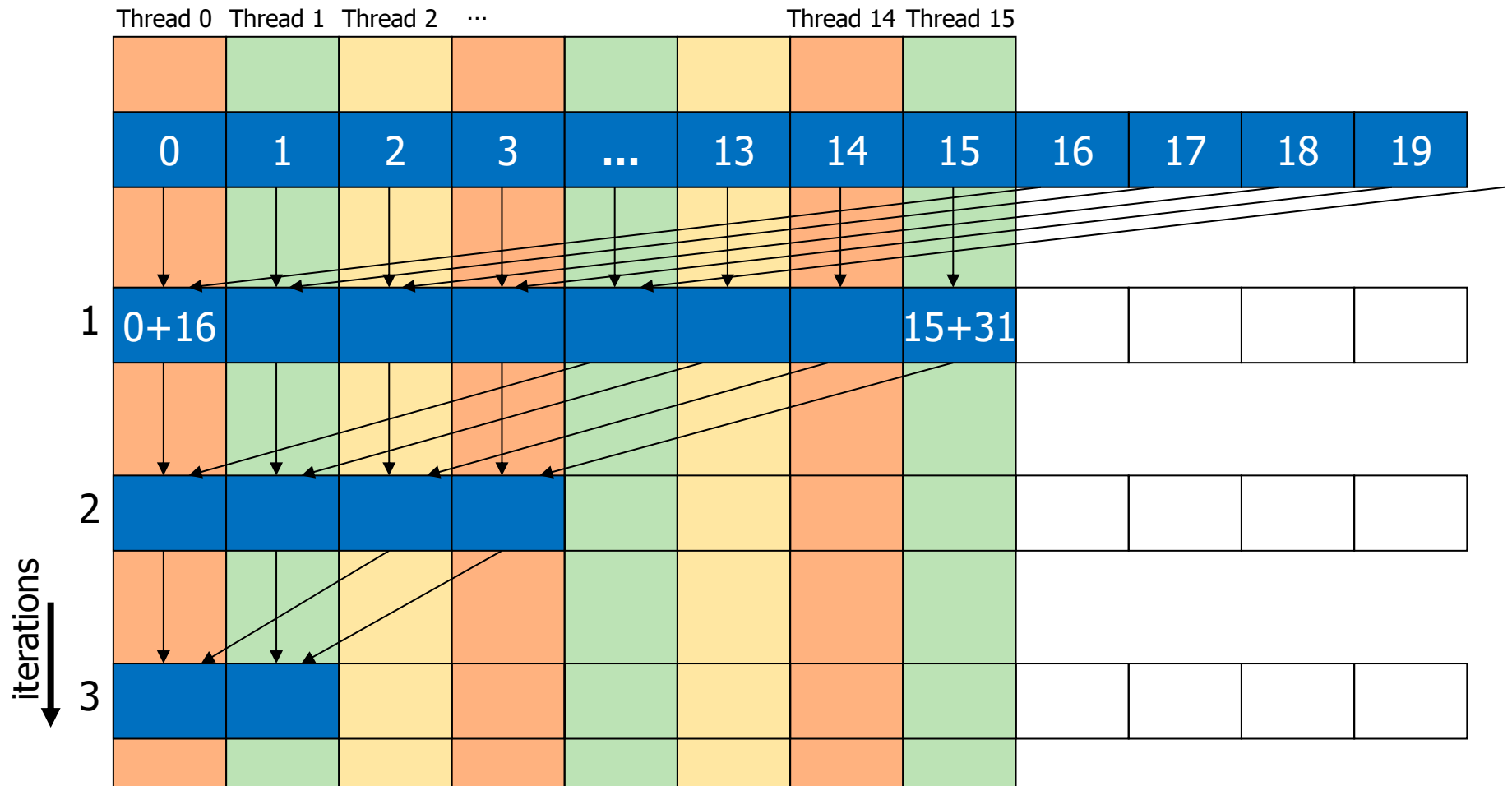
Parallel Patterns

Reduction Operation

- A **reduction** operation reduces a set of values to a single value
 - Sum, Product, Minimum, Maximum are examples
- **Properties of reduction**
 - Associativity
 - Commutativity
 - Identity value
- Reduction is a key primitive for parallel computing
 - E.g., MapReduce programming model

Divergence-Free Mapping (I)

- All active threads belong to the same warp

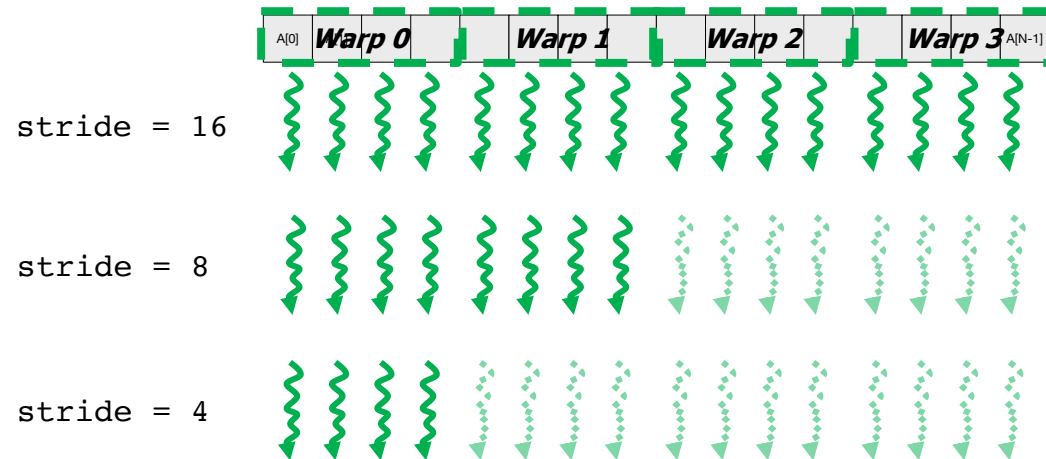


Divergence-Free Mapping (II)

■ Program with high SIMD utilization

```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0;  stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

Warp utilization
is maximized

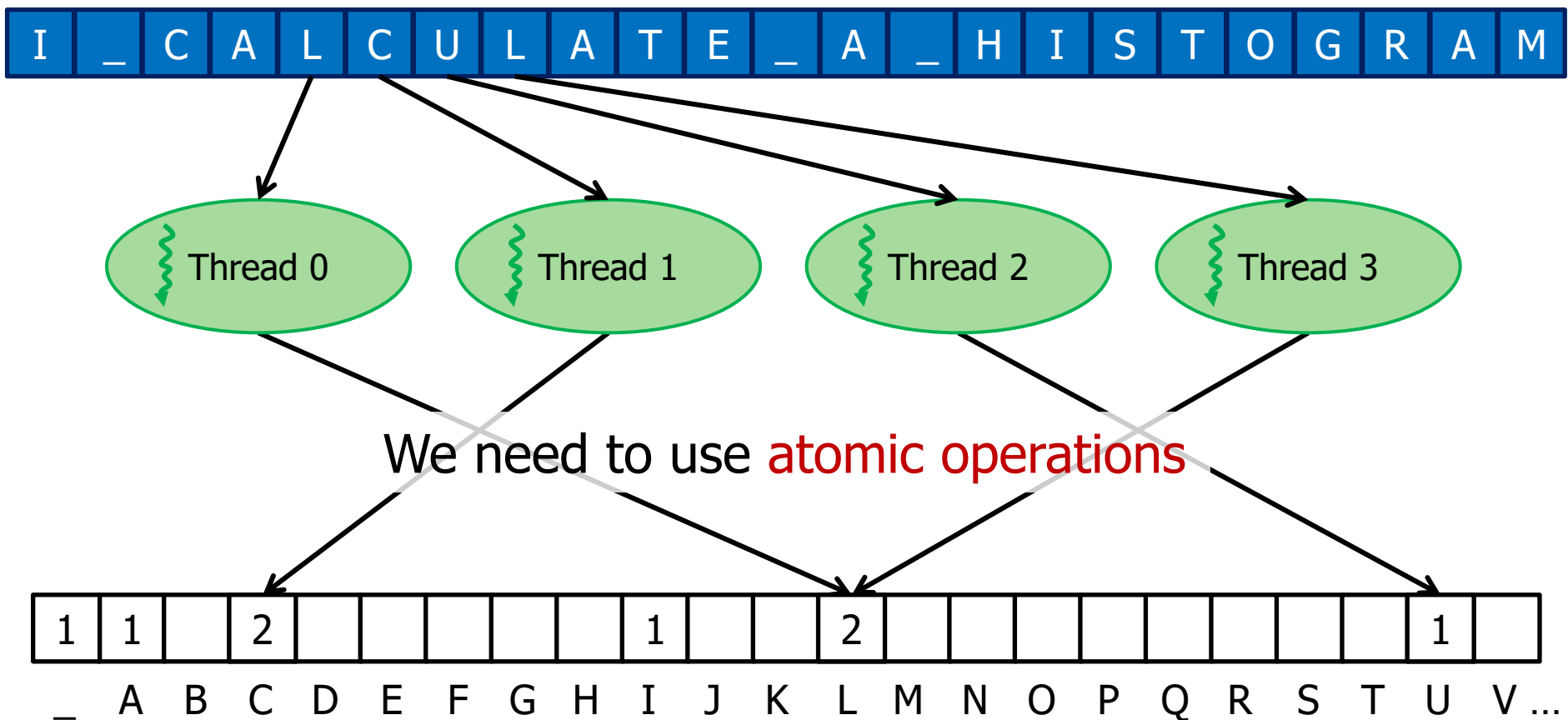


Histogram Computation

- Histogram is a frequently used computation for **reducing the dimensionality and extracting notable features** and patterns from large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlating heavenly object movements in astrophysics
 - ...
- Basic histograms - for **each element in the data set, use the value to identify a “bin” to increment**
 - Divide possible input value range into “bins”
 - Associate a counter to each bin
 - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

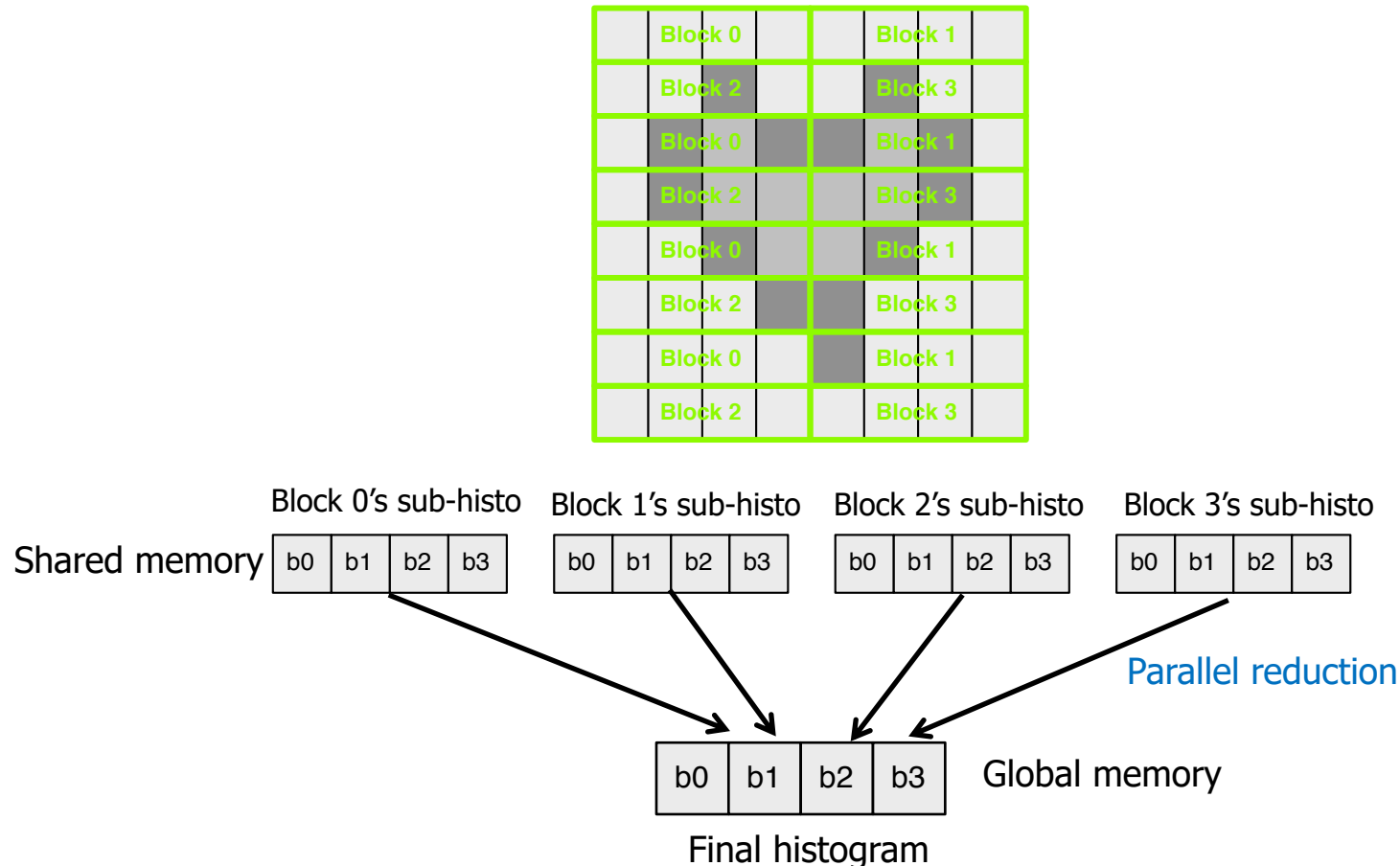
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
 - Each thread moves to element $\text{threadID} + \#\text{threads}$



Histogram Privatization

- **Privatization:** Per-block sub-histograms in shared memory
 - Threads use atomic operations in shared memory



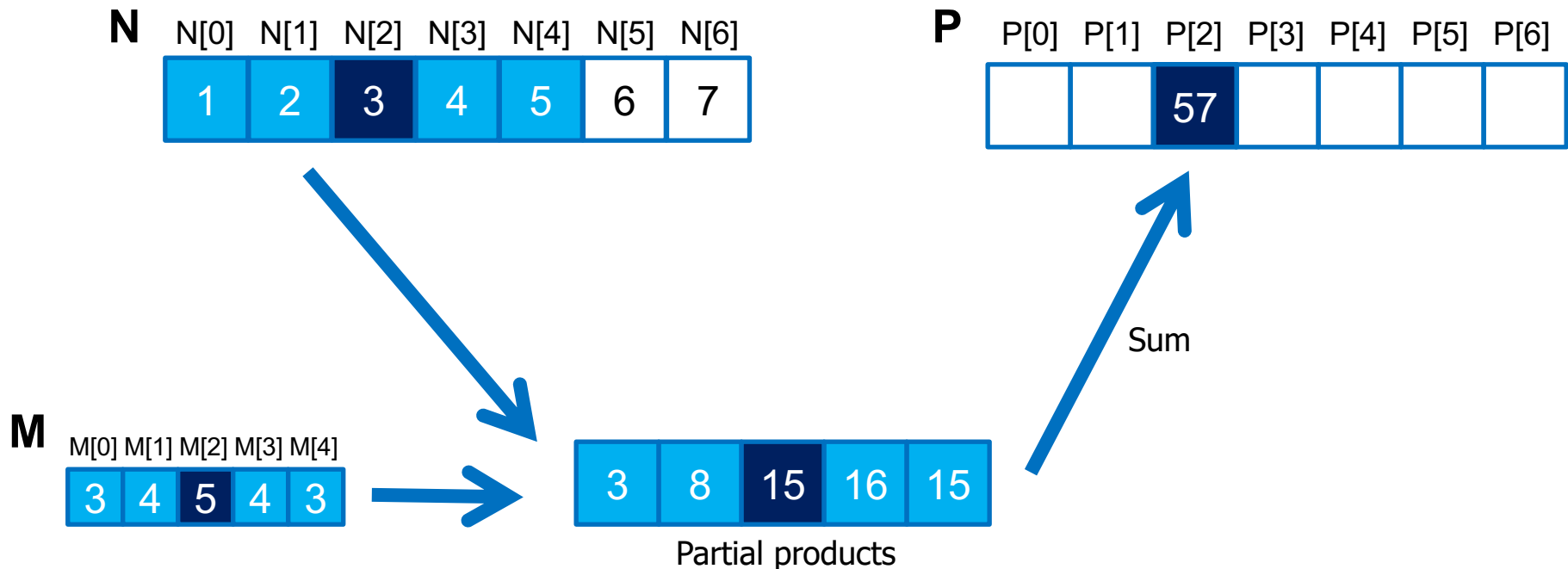
Convolution Applications

- **Convolution** is a widely-used operation in signal processing, image processing, video processing, and computer vision
- Convolution applies a **filter** or **mask** or **kernel*** on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a **weighted sum of a set of neighboring input elements**
 - Smoothing, sharpening, or blurring an image
 - Finding edges in an image
 - Removing noise, etc.
- Applications in machine learning and artificial intelligence
 - **Convolutional Neural Networks** (CNN or ConvNets)

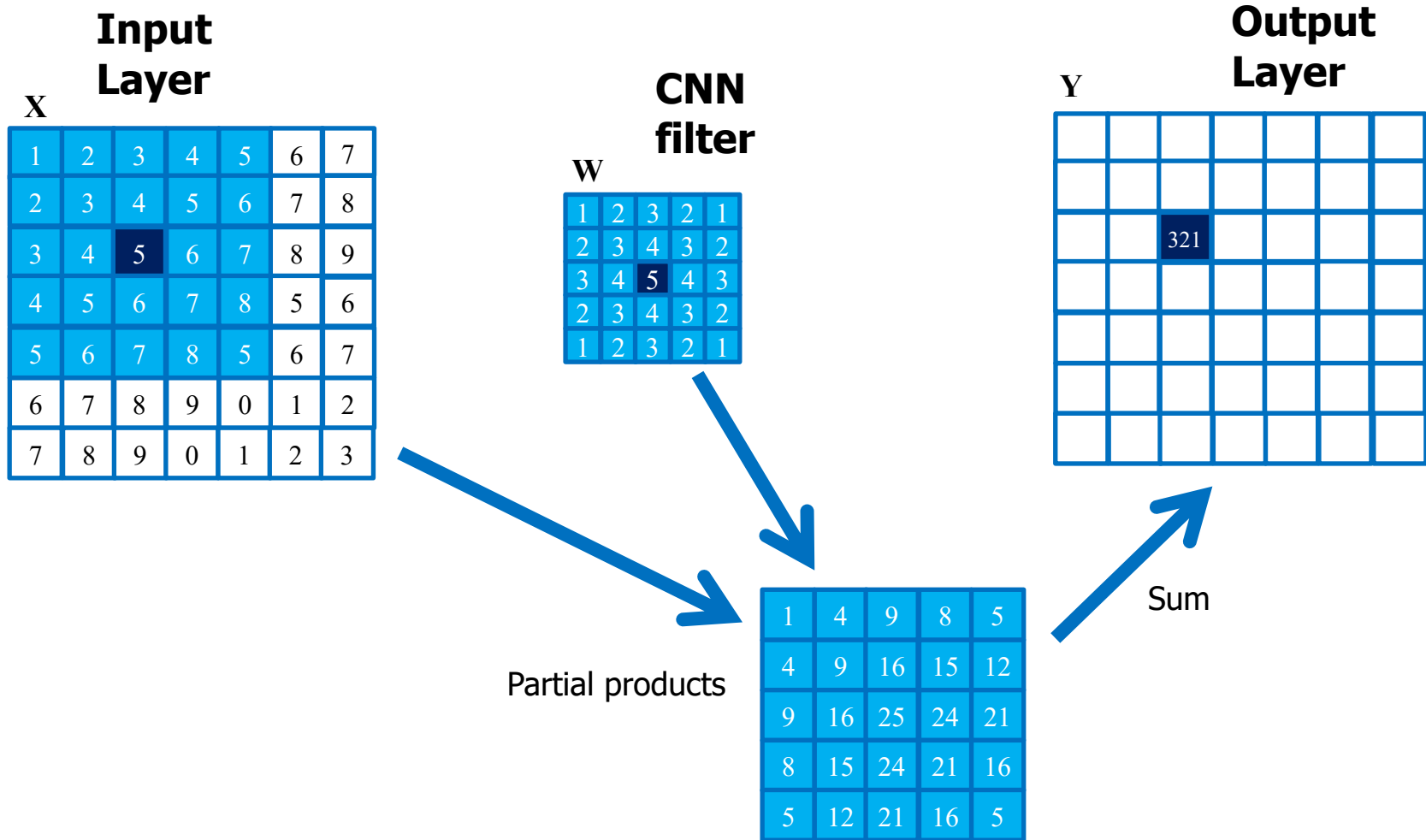
* The term “kernel” may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

1D Convolution Example

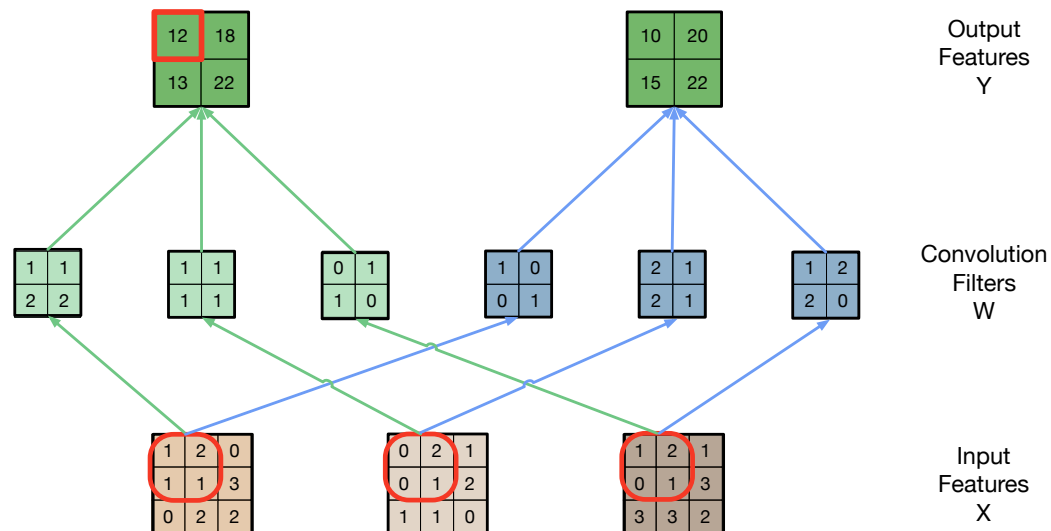
- Commonly used for audio processing
- Mask size is usually **an odd number of elements** for symmetry (5 in this example)
- Calculation of P[2]:



Another Example of 2D Convolution



Implementing a Convolutional Layer with Matrix Multiplication



$$\begin{array}{cccccccccccc}
 1 & 1 & 2 & 2 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
 1 & 0 & 0 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 0
 \end{array}
 *
 \begin{array}{c}
 1 \\
 2 \\
 1 \\
 1 \\
 3 \\
 0 \\
 2 \\
 2 \\
 1 \\
 1 \\
 0 \\
 1 \\
 1 \\
 2 \\
 1 \\
 0 \\
 1 \\
 3 \\
 3 \\
 3 \\
 2
 \end{array}
 =
 \begin{array}{cccc}
 12 & 18 & 13 & 22 \\
 10 & 20 & 15 & 22
 \end{array}$$

Convolution Filters W'

Input Features X (unrolled)

Output Features Y

Prefix Sum (Scan)

- **Prefix sum** or **scan** is an operation that takes an input array and an associative operator,
 - E.g., addition, multiplication, maximum, minimum
- And returns an output array that is the result of recursively applying the associative operator on the elements of the input array
- Input array $[x_0, x_1, \dots, x_{n-1}]$
- Associative operator \oplus
- An output array $[y_0, y_1, \dots, y_{n-1}]$ where
 - **Exclusive** scan: $y_i = x_0 \oplus x_1 \oplus \dots \oplus x_{i-1}$
 - **Inclusive** scan: $y_i = x_0 \oplus x_1 \oplus \dots \oplus x_i$

Hierarchical (Inclusive) Scan

Input	<i>Block 0</i>				<i>Block 1</i>				<i>Block 2</i>				<i>Block 3</i>			
	1	2	3	4	1	1	1	1	0	1	2	3	2	2	2	2

Per-block (Inclusive) Scan

1	3	6	10	1	2	3	4	0	1	3	6	2	4	6	8
---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

10	4	6	8
----	---	---	---

Scan Partial Sums

10	14	20	28
----	----	----	----

Inter-block synchronization

- Kernel termination and
 - Scan on CPU, or
 - Launch new scan kernel on GPU
- Atomic operations in global memory

Add

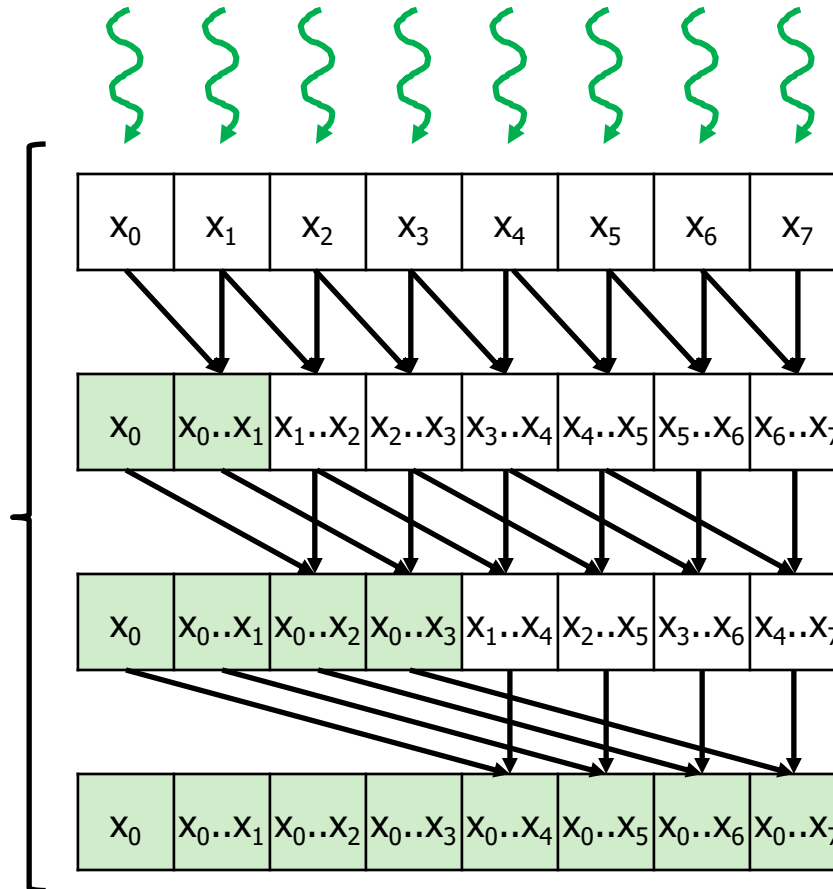
1	3	6	10	1	2	3	4	0	1	3	6	2	4	6	8
---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

Output (Inclusive Scan)

1	3	6	10	11	12	13	14	14	15	17	20	22	24	26	28
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

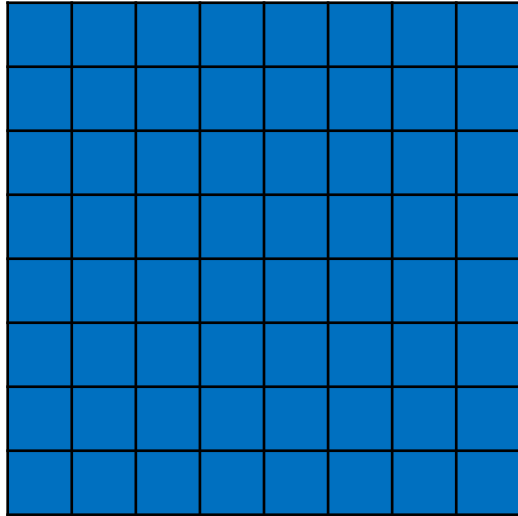
Kogge-Stone Parallel (Inclusive) Scan

Observation:
memory locations
are reused

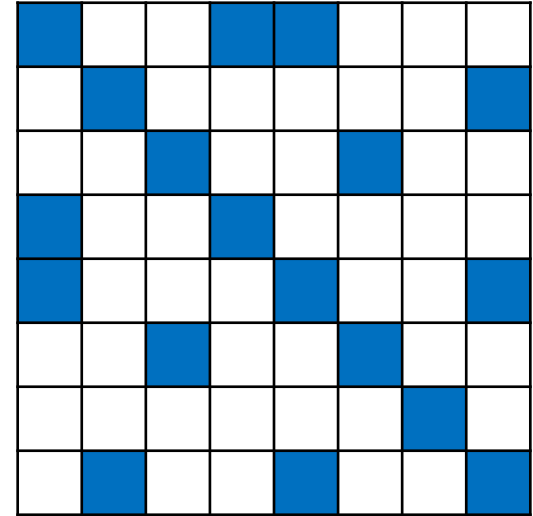


Sparse Matrices

A **dense matrix** is one where the majority of elements are not zero



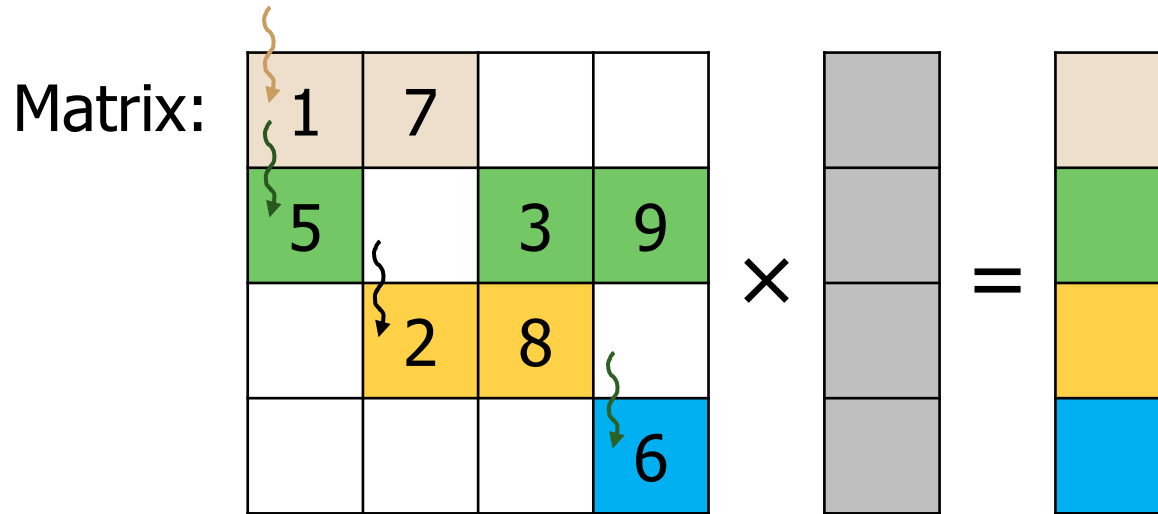
A **sparse matrix** is one where many elements are zero
(many real world systems are sparse)



■ Opportunities:

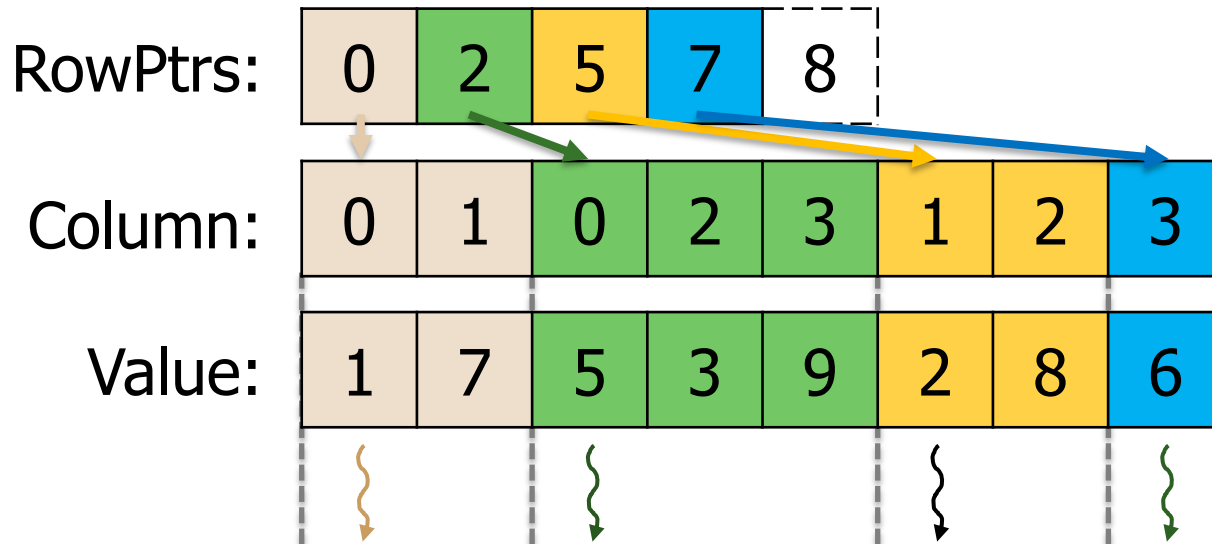
- ❑ Do not need to allocate **space for zeros** (save memory capacity)
- ❑ Do not need to **load zeros** (save memory bandwidth)
- ❑ Do not need to **compute with zeros** (save computation time)

SpMV/CSR



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element



Graph Search

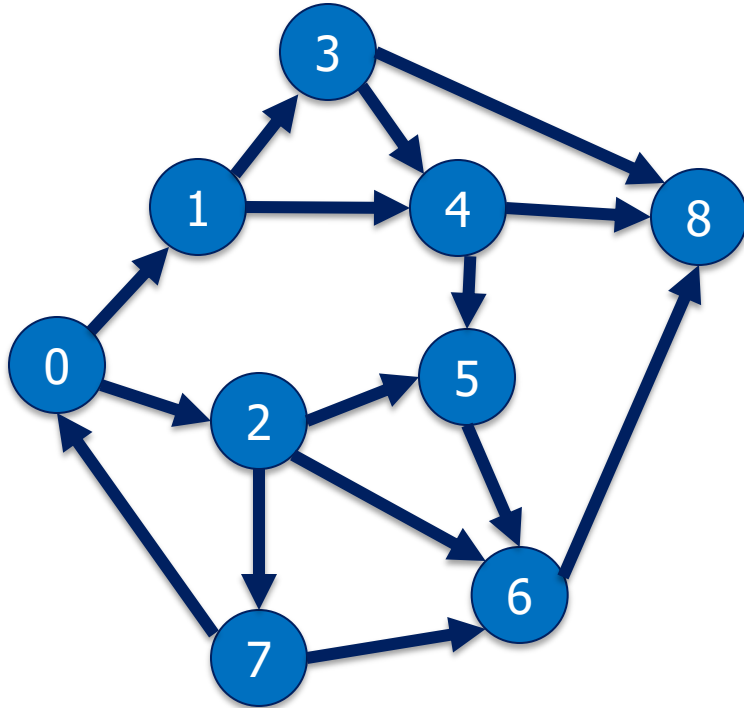
Dynamic Data Extraction

- The data to be processed **in each phase of computation need to be dynamically determined and extracted** from a bulk data structure
 - Harder when the bulk data structure is not organized for massively parallel access, such as graphs
- **Graph algorithms** are popular examples that perform dynamic data extraction
 - Widely used in EDA, NLZP, and large scale optimization applications
 - We will use **Breadth-First Search** (BFS) as an example

Main Challenges of Dynamic Data Extraction

- Input data need to be organized for **locality, coalescing, and contention avoidance** as they are extracted during execution
- The amount of **work and level of parallelism** often grow and **shrink** during execution
 - ❑ As more or less data is extracted during each phase
 - ❑ Hard to efficiently fit into one GPU kernel configuration, without dynamic parallelism support (Kepler and beyond)
 - ❑ Different kernel strategies fit different data sizes

Graph and Sparse Matrix are Closely Related

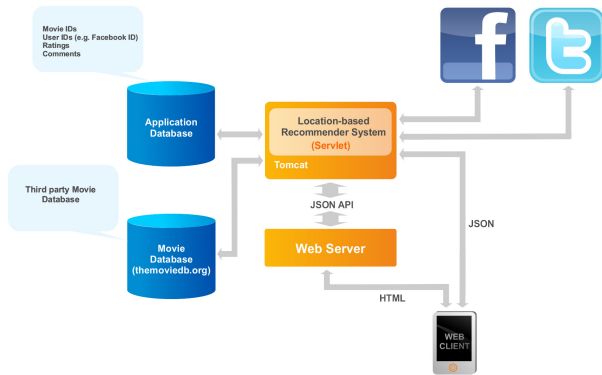


	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									

Adjacency matrix

Recall: Sparse Matrices are Widespread Today

Recommender Systems



- Collaborative Filtering

Graph Analytics



- PageRank
- Breadth First Search
- Betweenness Centrality

Neural Networks



- Sparse DNNs
- Graph Neural Networks

Recall: Compressed Sparse Row (CSR)

Matrix:

1	7		
5		3	9
	2	8	
			6

Store nonzeros of the
same row adjacently
and an index to the
first element of each
row

RowPtrs:

0	2	5	7	8
---	---	---	---	---

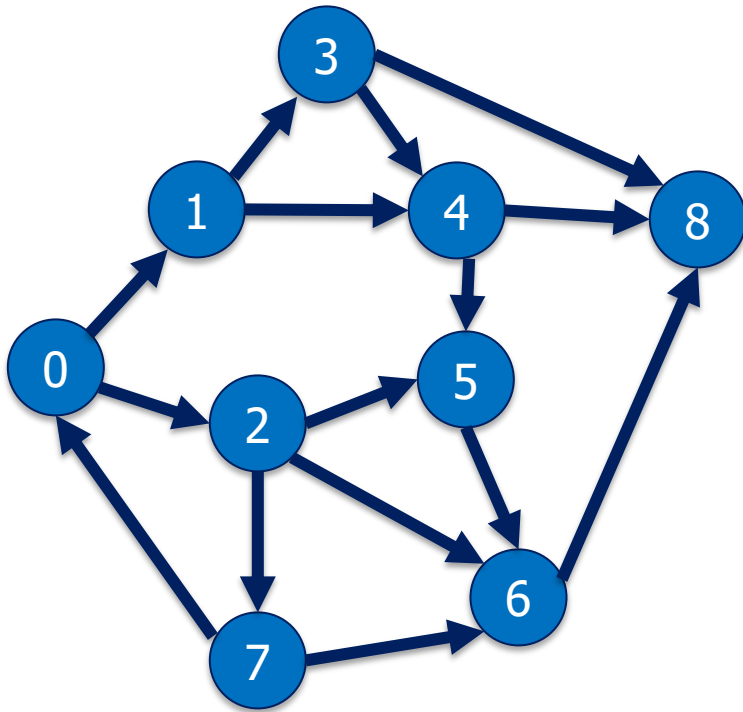
Column:

0	1	0	2	3	1	2	3
---	---	---	---	---	---	---	---

Value:

1	7	5	3	9	2	8	6
---	---	---	---	---	---	---	---

(Compressed) Edge Representation of a Graph

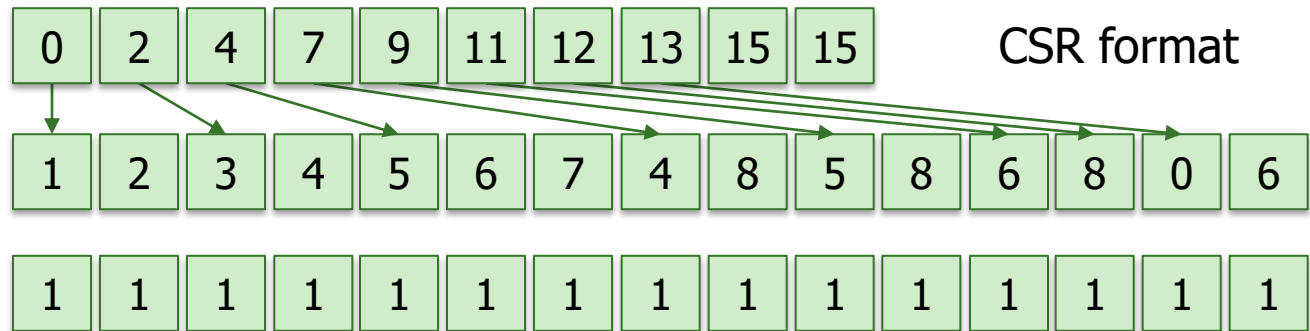


	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									

row pointers
source[10]

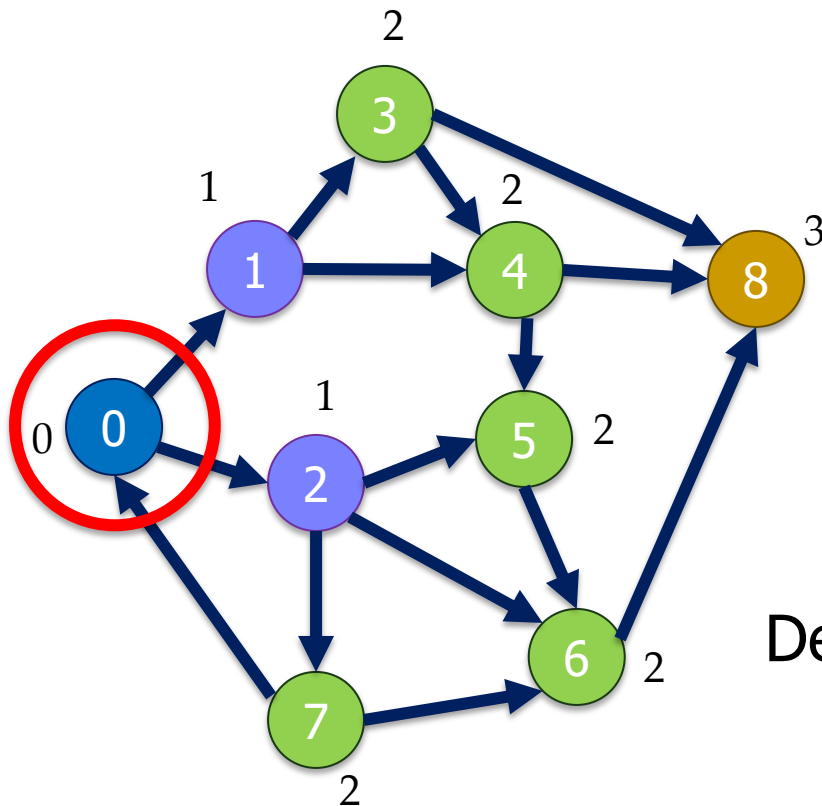
column indices
destination[15]

non-zero elements
data[15]



Breadth-First Search (BFS)

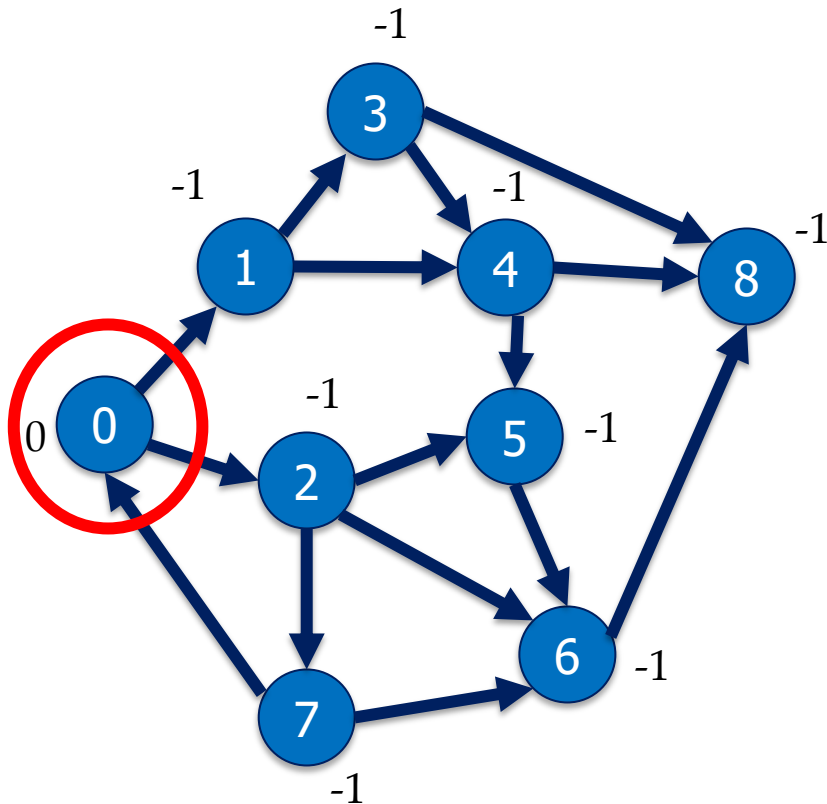
- To determine the **minimal number of hops** that is required **to go from a source node to a destination node** (or all destinations)



Desirable Outcome

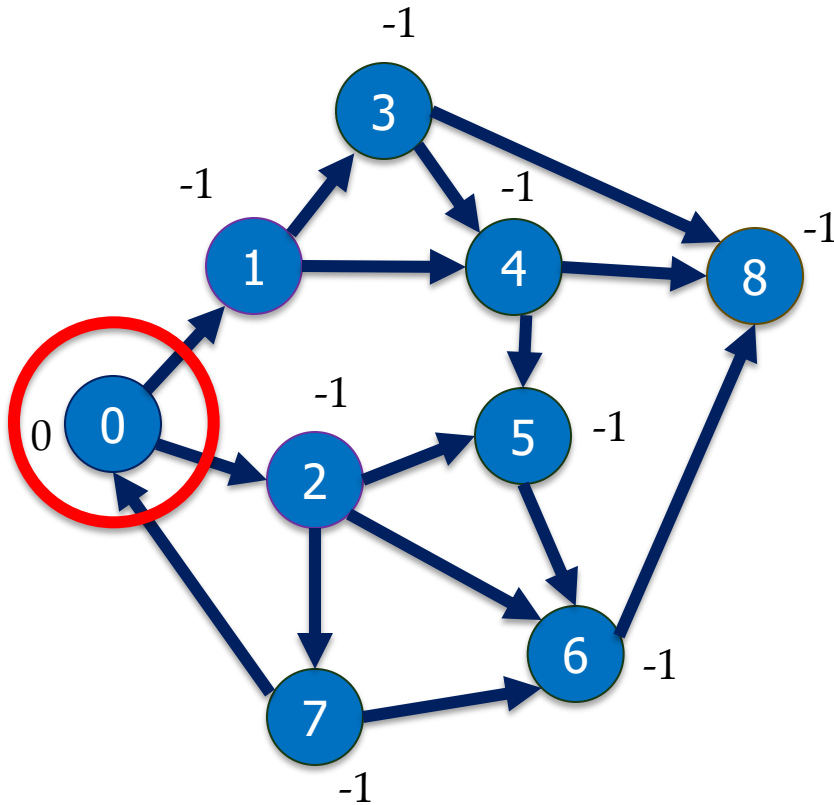
Breadth-First Search: Example

- Start with a source node
- Identify and mark all nodes that can be reached from the source node with 1 hop, 2 hops, 3 hops, ...



Initial Condition

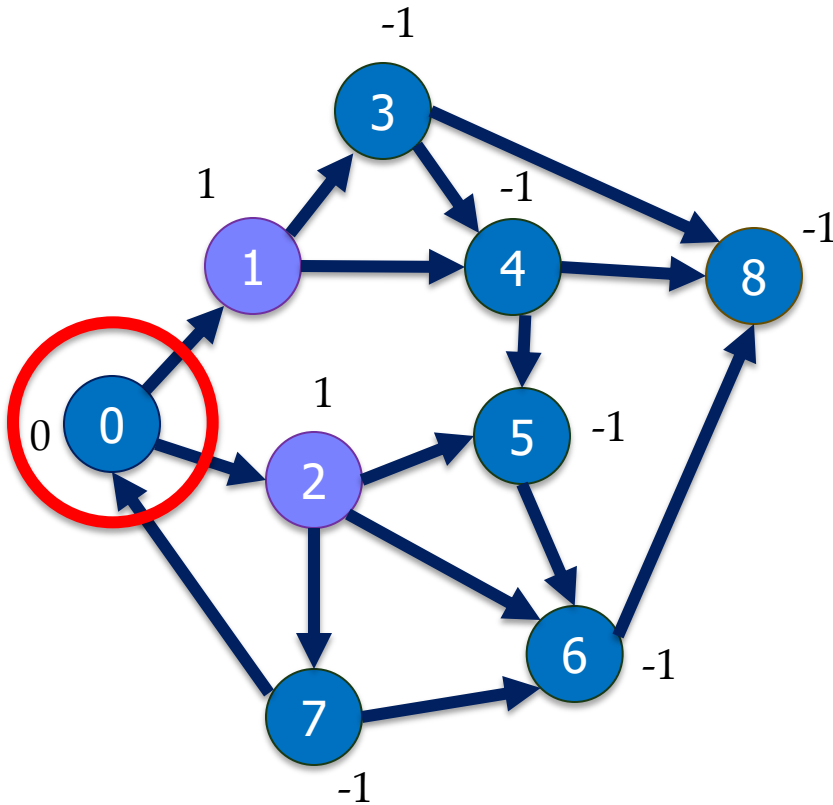
Breadth-First Search – Initial Condition



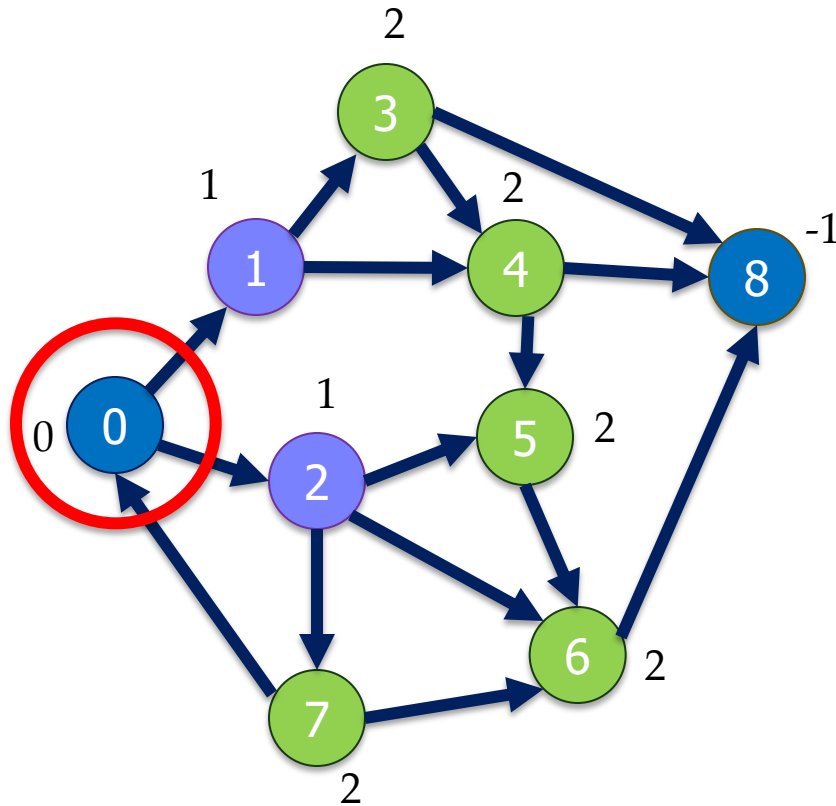
Breadth-First Search – 1 Hop

- First Frontier (level 1 nodes)

□ 1, 2

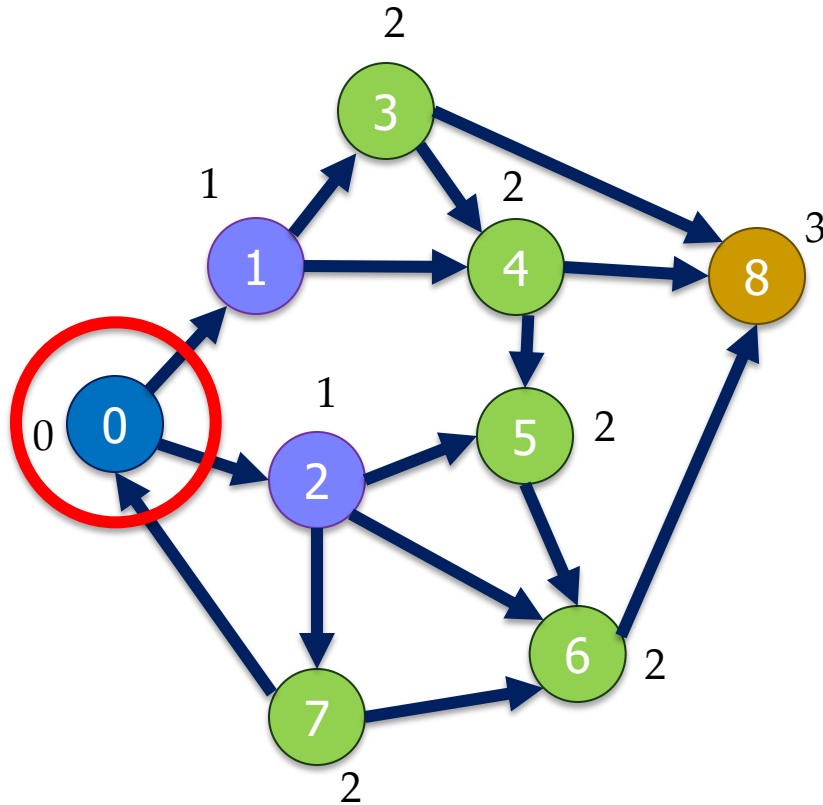


Breadth-First Search – 2 Hops



- First Frontier (level 1 nodes)
 - 1, 2
- Second frontier (level 2 nodes)
 - 3, 4, 5, 6, 7

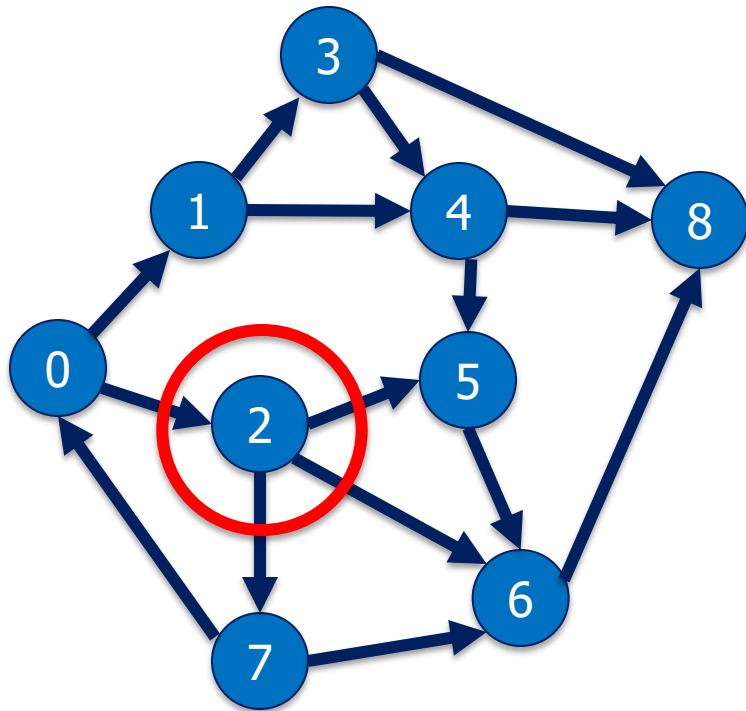
Breadth-First Search – 3 Hops



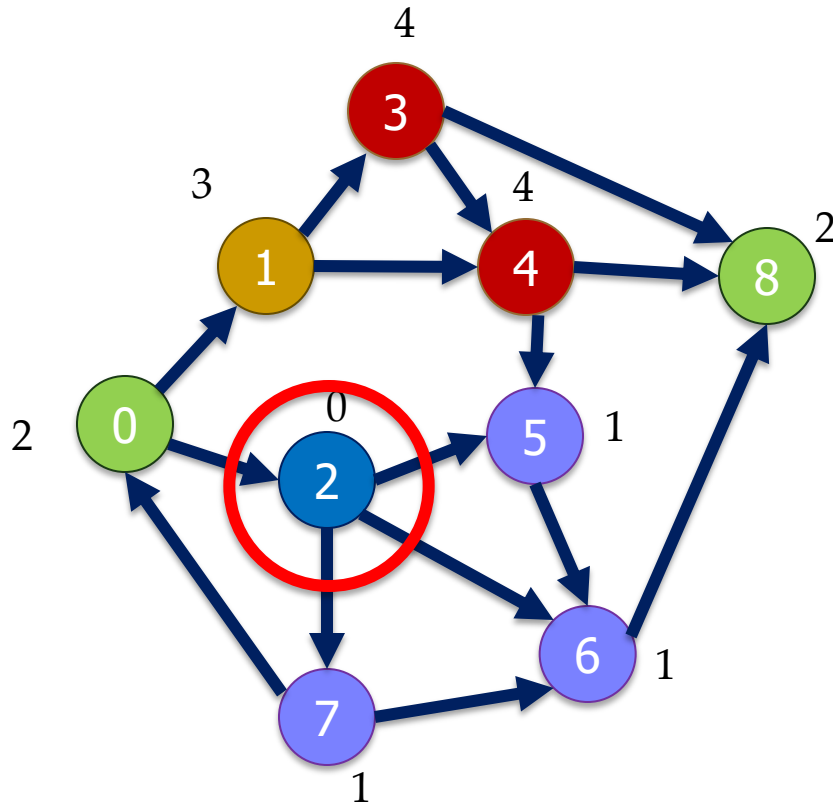
Desirable Outcome

- First Frontier (level 1 nodes)
 - 1, 2
- Second frontier (level 2 nodes)
 - 3, 4, 5, 6, 7
- Third frontier (level 3 nodes)
 - 8
- ...

Breadth-First Search – Node 2 as Source

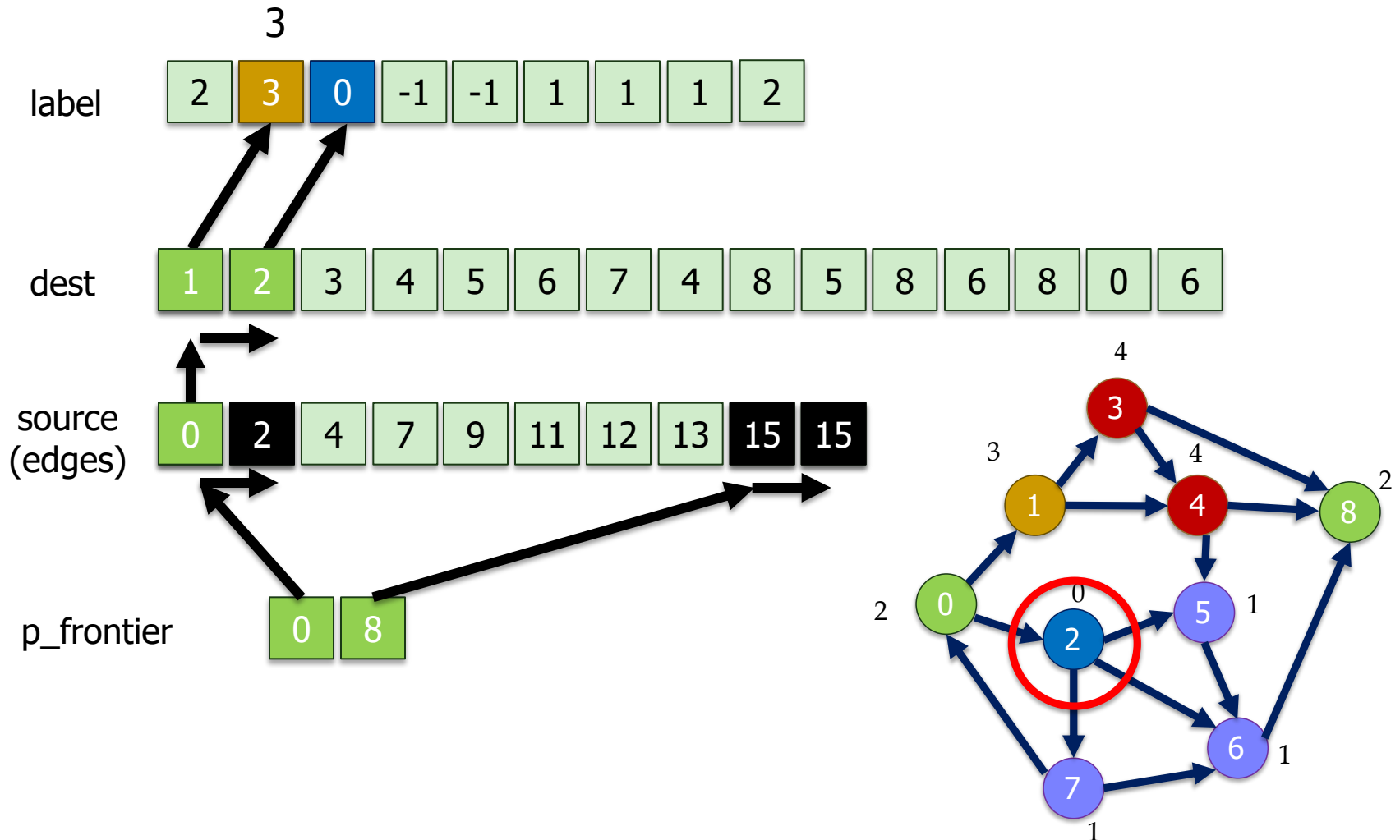


Breadth-First Search – Node 2 as Source



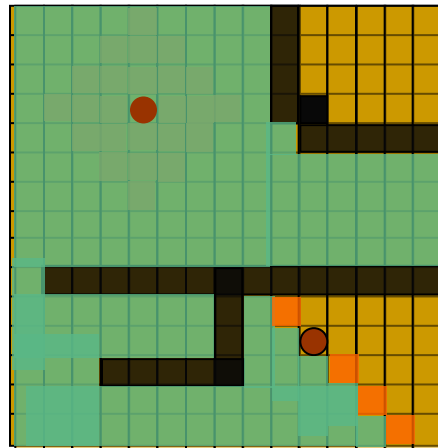
- First Frontier (level 1 nodes)
 - 5, 6, 7
- Second frontier (level 2 nodes)
 - 0, 8
- Third frontier (level 3 nodes)
 - 1
- ...

BFS: Processing the Frontier (2nd Iteration)

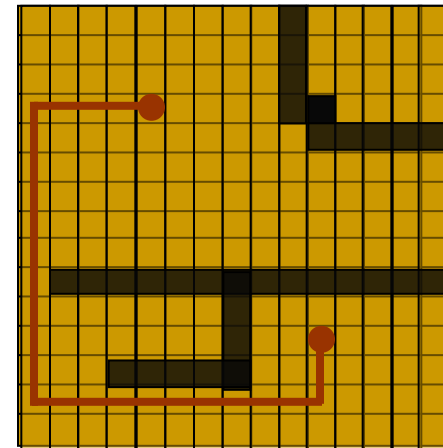


BFS Use Example in VLSI CAD

■ Maze Routing

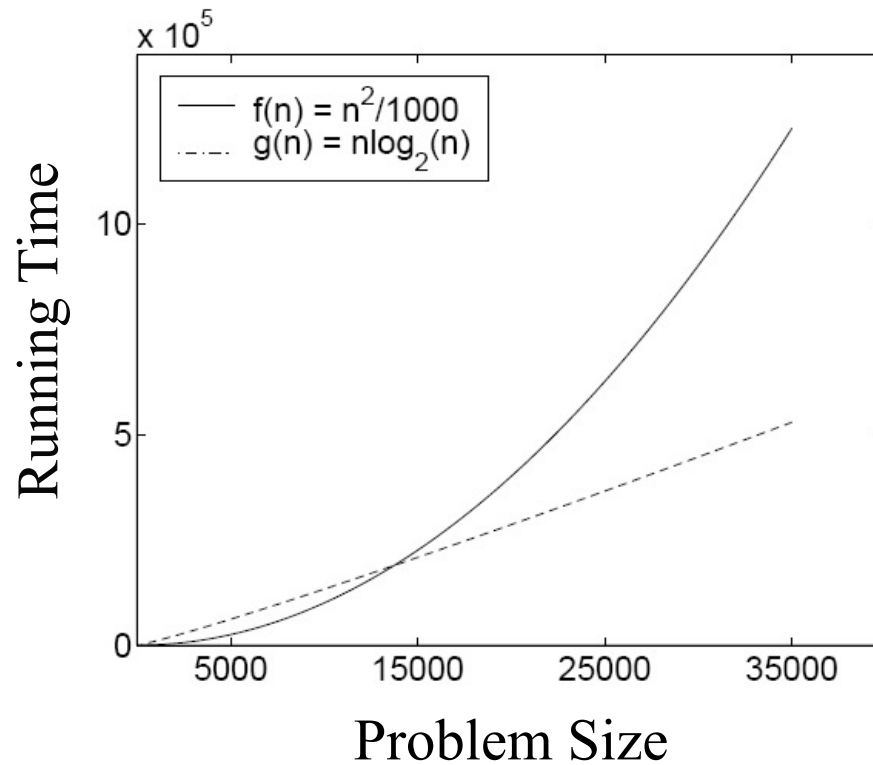


● net terminal
■ blockage



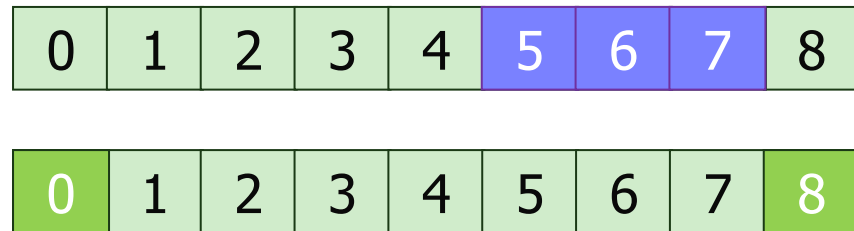
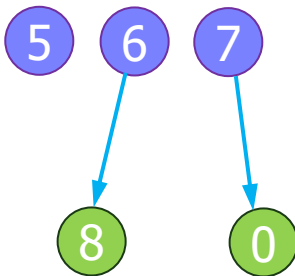
Potential Pitfall of Parallel Algorithms

- Greatly accelerated n^2 algorithm is still slower than an $n\log(n)$ algorithm for large data sets
- Always need to keep an eye on fast sequential algorithm as the baseline



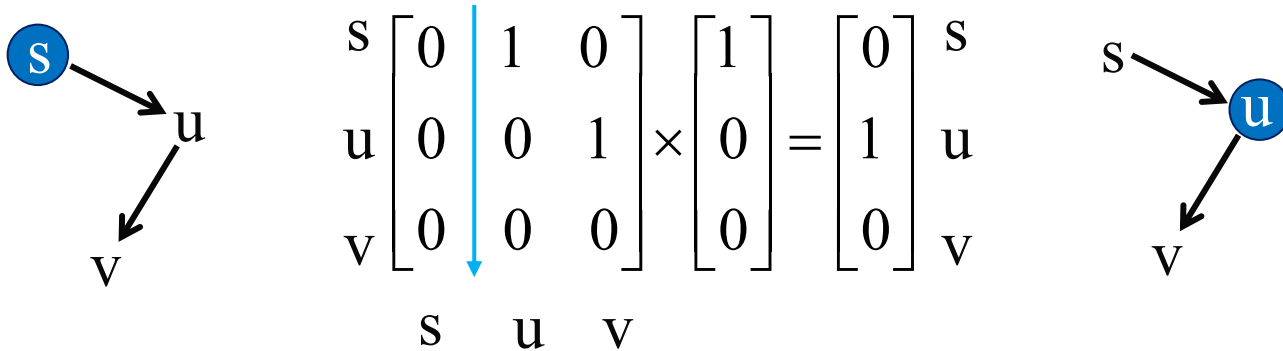
Node-Oriented Parallelization

- Each thread is dedicated to one node
 - All nodes visited in all iterations
 - Every thread examines neighbor nodes to determine if its node will be a frontier node in the next phase
 - Complexity $O(VL+E)$ (Compared with $O(V+E)$)
 - L is the number of levels
 - Slower than the sequential version for large graphs
 - Especially for sparsely connect graphs



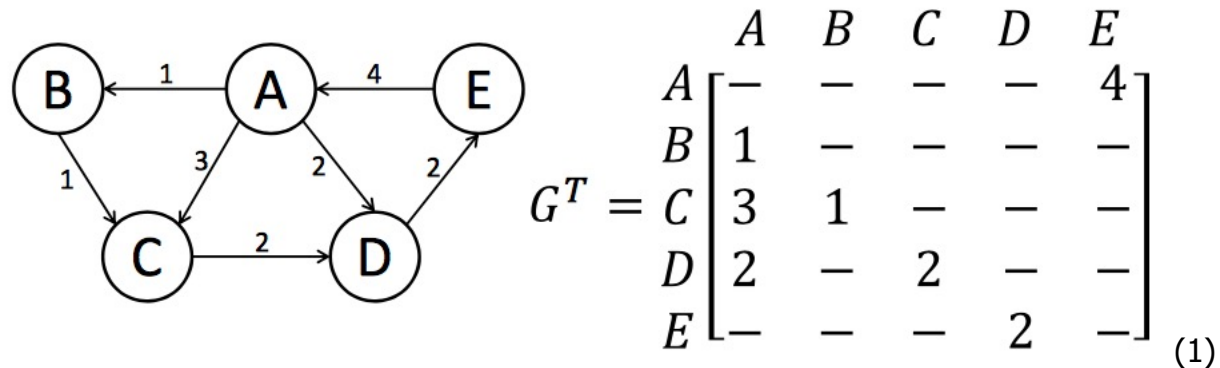
Matrix-Based Parallelization

- Propagation is done through matrix-vector multiplication
 - For sparsely connected graphs, the connectivity matrix will be a sparse matrix
- Complexity $O(V+EL)$ (compared with $O(V+E)$)
 - Slower than sequential for large graphs



Linear Algebraic Formulation

■ Logical representation and adjacency matrix



■ Vertex programming model

GraphMat Processing Model	
1	For each Vertex V
2	For each incoming edge E(U,V) from active vertex U
3	Res \leftarrow Process_Edge (E_weight, U_prop, [OPTIONAL]V_prop)
4	V_temp \leftarrow Reduce (V_temp, Res)
5	End
6	End
7	For each Vertex V,
8	V_prop \leftarrow Apply (V_temp, V_prop, V_const)
9	End

Fig. 1: Simplified GraphMat processing model. Note that this is slightly different from the original GraphMat [46] in that it integrates Send_Message with Apply. (2)

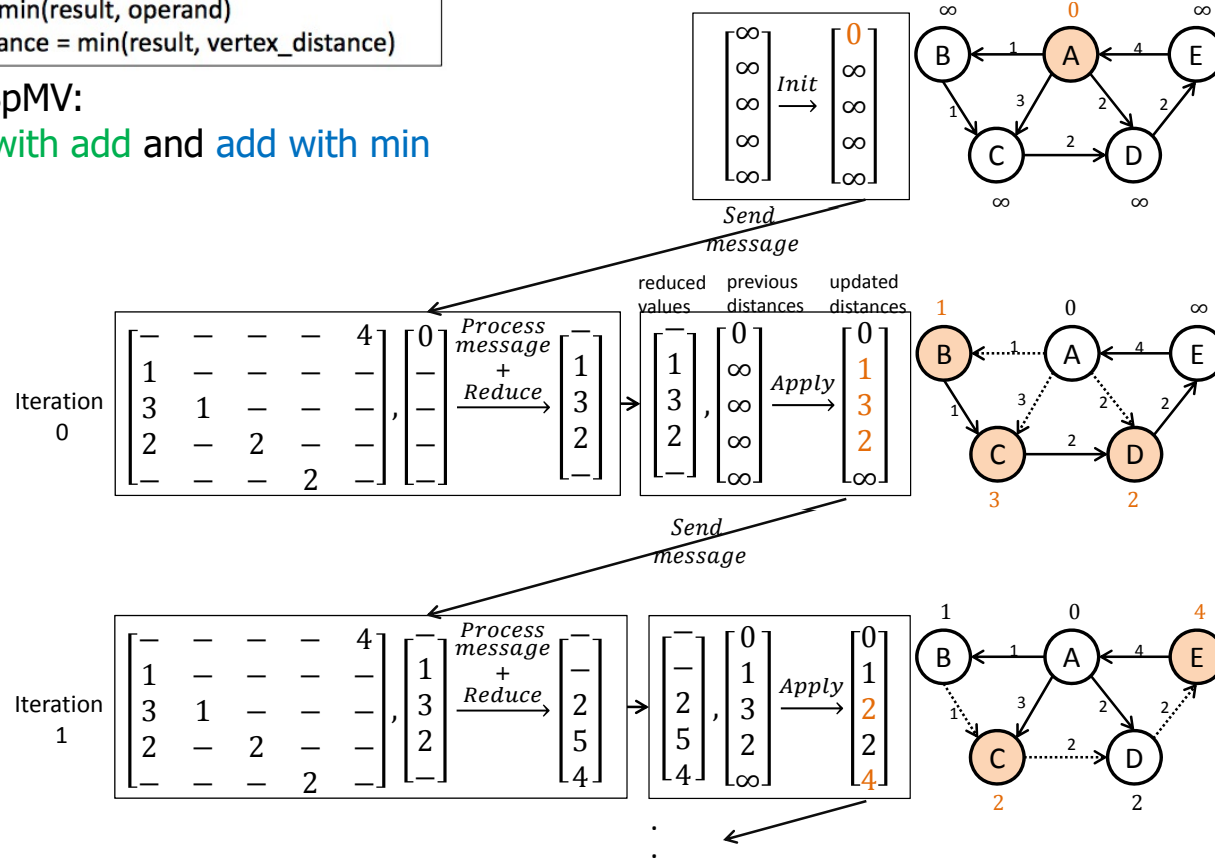
Mapping Vertex Programs to SpMV

■ Example: Single Source Shortest Path (SSSP)

SEND_MESSAGE : $\text{message} := \text{vertex_distance}$
PROCESS_MESSAGE : $\text{result} := \text{message} + \text{edge_value}$
REDUCE : $\text{result} := \min(\text{result}, \text{operand})$
APPLY : $\text{vertex_distance} = \min(\text{result}, \text{vertex_distance})$

Generalized SpMV:

Replace **mul** with **add** and **add** with **min**

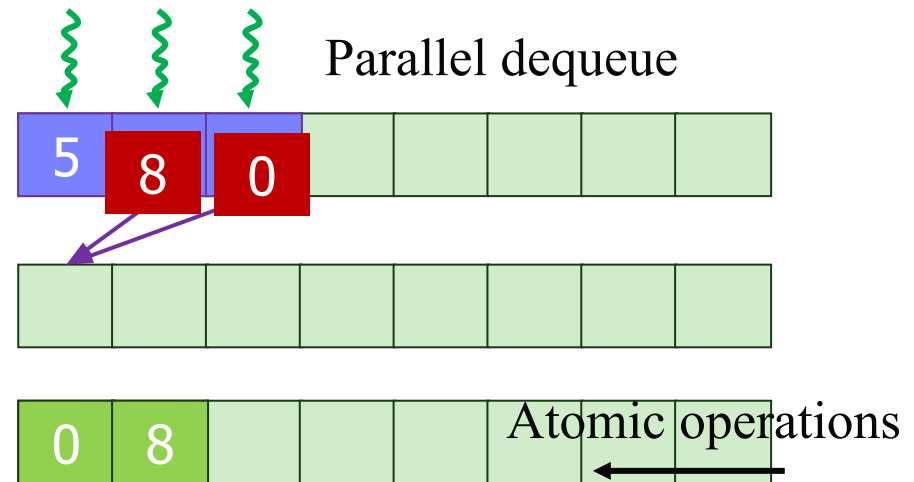
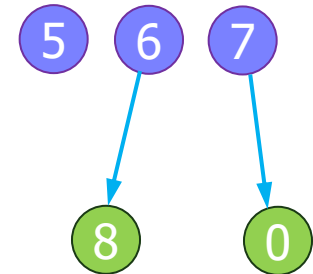


Need a More General Technique

- To efficiently handle most graph types
- Use more specialized formulation when appropriate as an optimization
- Efficient **queue-based parallel algorithms**
 - Hierarchical scalable queue implementation
 - Hierarchical kernel arrangements

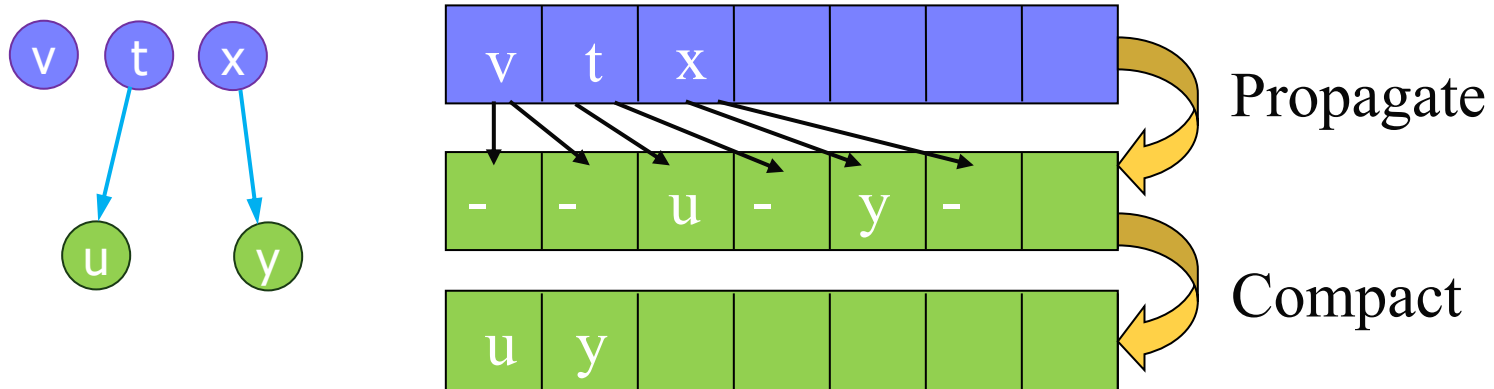
An Initial Attempt

- Manage the queue structure
 - ❑ Complexity: $O(V+E)$
 - ❑ Dequeue in parallel
 - ❑ Each frontier node is a thread
 - ❑ Enqueue in sequence using **atomic operations**
 - Poor coalescing
 - Poor scalability
 - ❑ No speedup

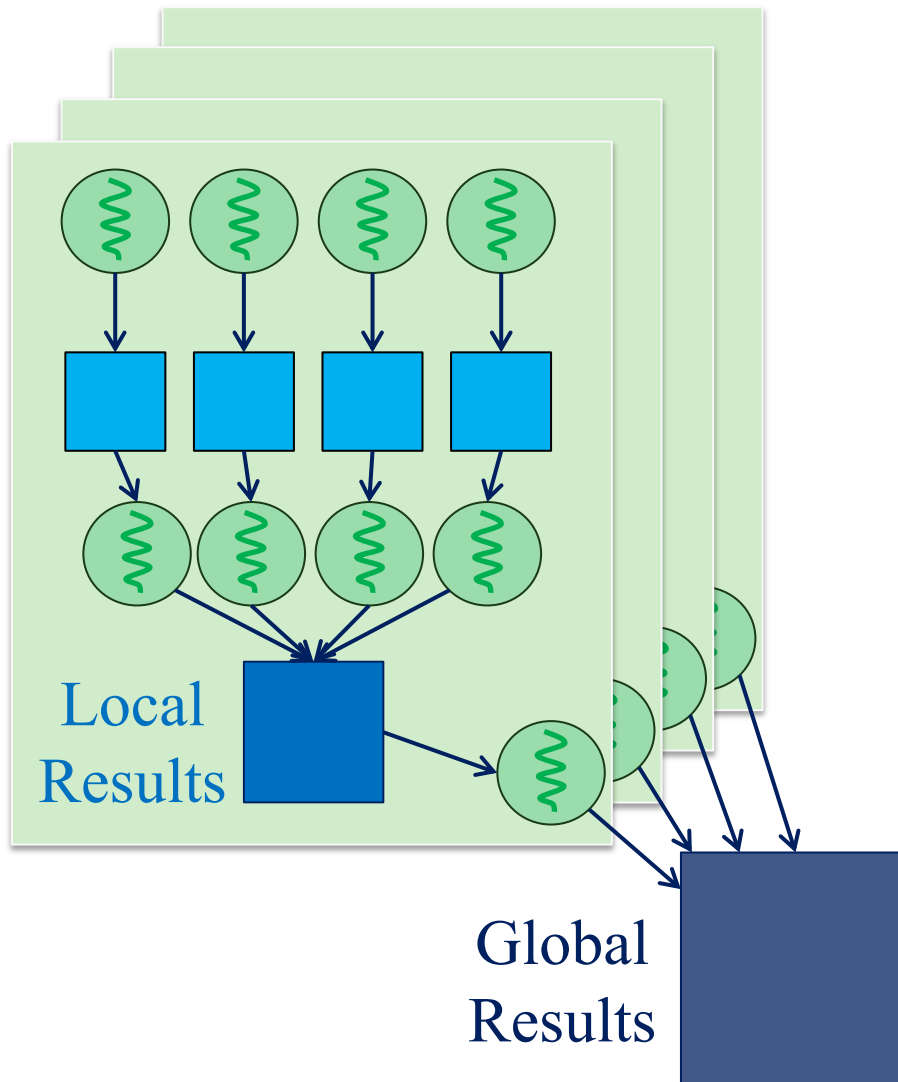


Parallel Insert-Compact Queues

- Parallel enqueue with **compaction** cost
- Not suitable for light-node problems



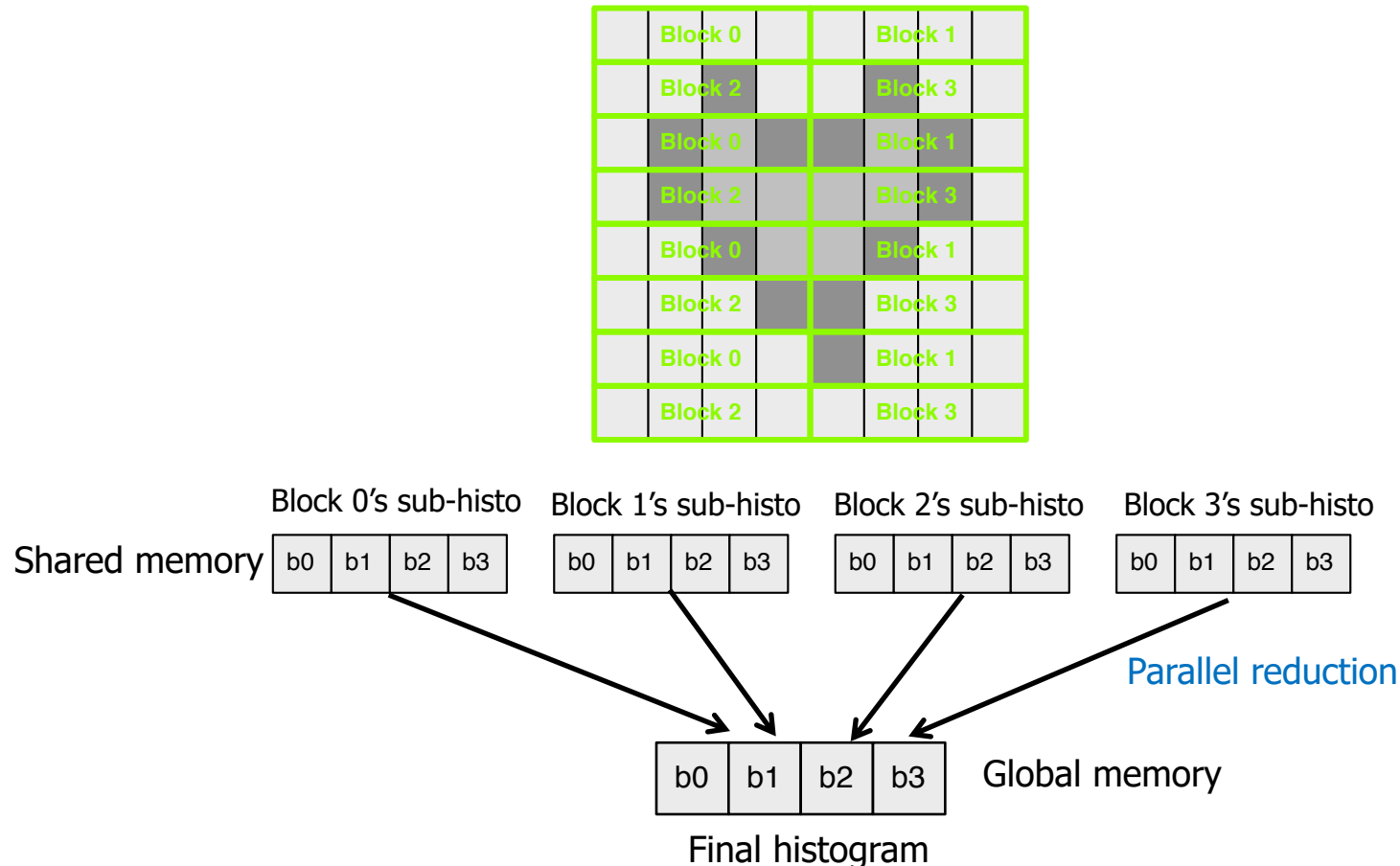
(Output) Privatization



- Avoid contention by aggregating updates locally
- Requires storage resources to keep copies of data structures

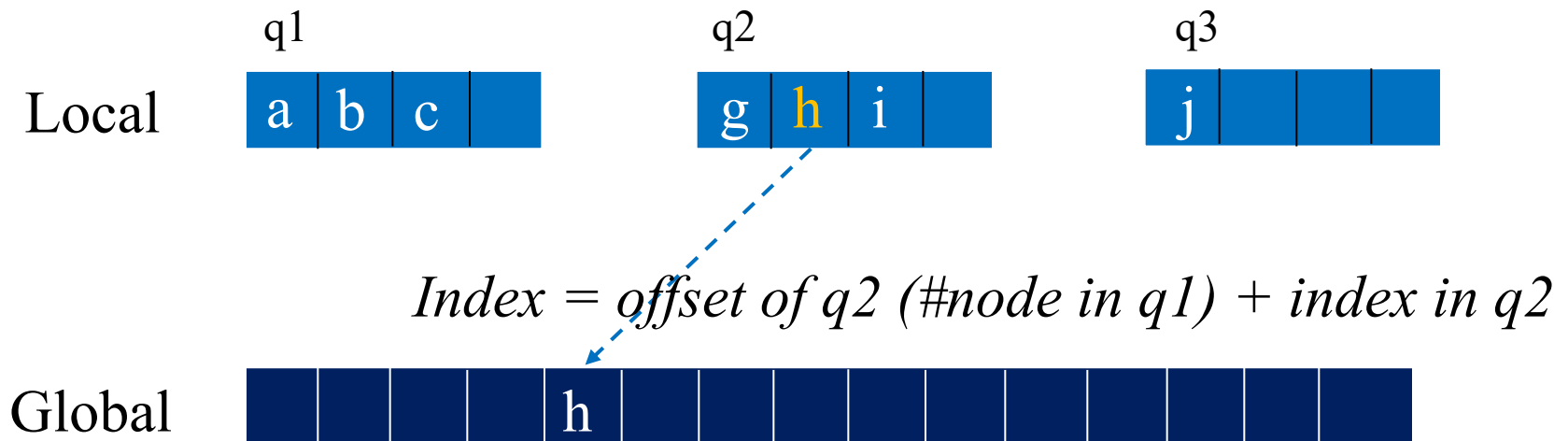
Recall: Histogram Privatization

- **Privatization:** Per-block sub-histograms in shared memory
 - Threads use atomic operations in shared memory

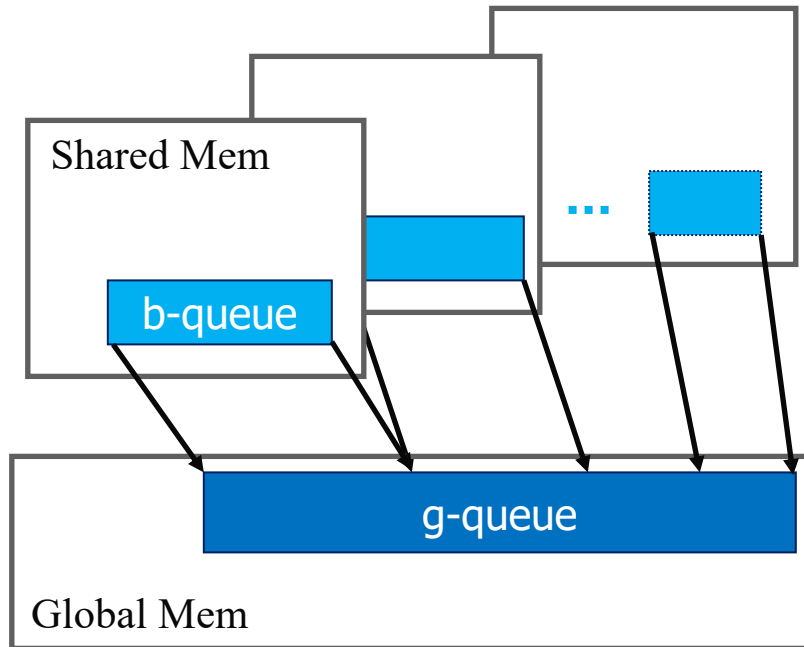


Basic Ideas

- Each thread processes one or more frontier nodes and inserts new frontier nodes into its private queues
- Find a location in the global queue for each new frontier node
- Build queue of next frontier hierarchically



Two-level Hierarchy



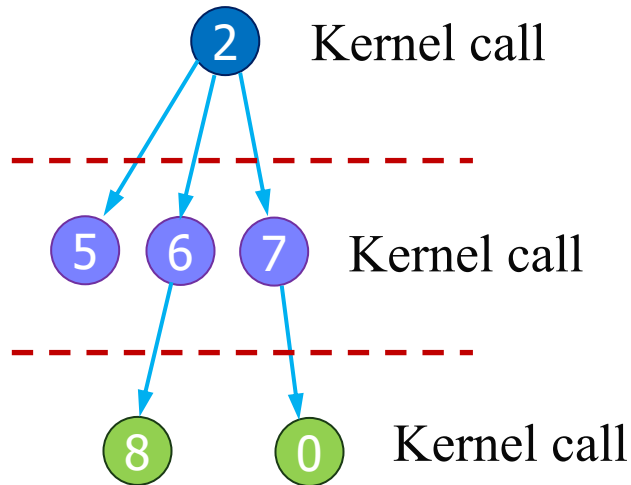
- **Block queue (b-queue)**
 - ❑ Inserted by all threads in a block
 - ❑ Resides in Shared Memory
- **Global queue (g-queue)**
 - ❑ Inserted only when a block completes
- **Problem:**
 - ❑ Collision on b-queues
 - ❑ Threads in the same block can cause heavy contention

Hierarchical Queue Management

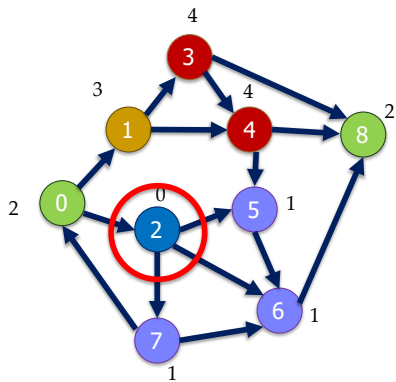
- **Advantage** and **limitation**

- The technique can be applied to any inherently sequential data structure
- As long as the **exact global ordering between queue contents is not required** for correctness or optimality (more of a list)
- The b-queues are limited by the capacity of shared memory
 - If we know the upper limit of the degree, we can adjust the number of threads per block accordingly
 - Overflow mechanism to ensure robustness

Kernel Arrangement



- Creating global barriers needs frequent kernel launches
- Too much overhead
- Solutions:
 - ❑ Partially use GPU-synchronization
 - ❑ Multi-layer Kernel Arrangement
 - ❑ Dynamic Parallelism
 - ❑ Persistent threads with global barriers



Hierarchical Kernel Arrangement

- Customize kernels based on the size of frontiers
- Use fast barrier synchronization when the frontier is small



Kernel 1: Intra-block synchronization



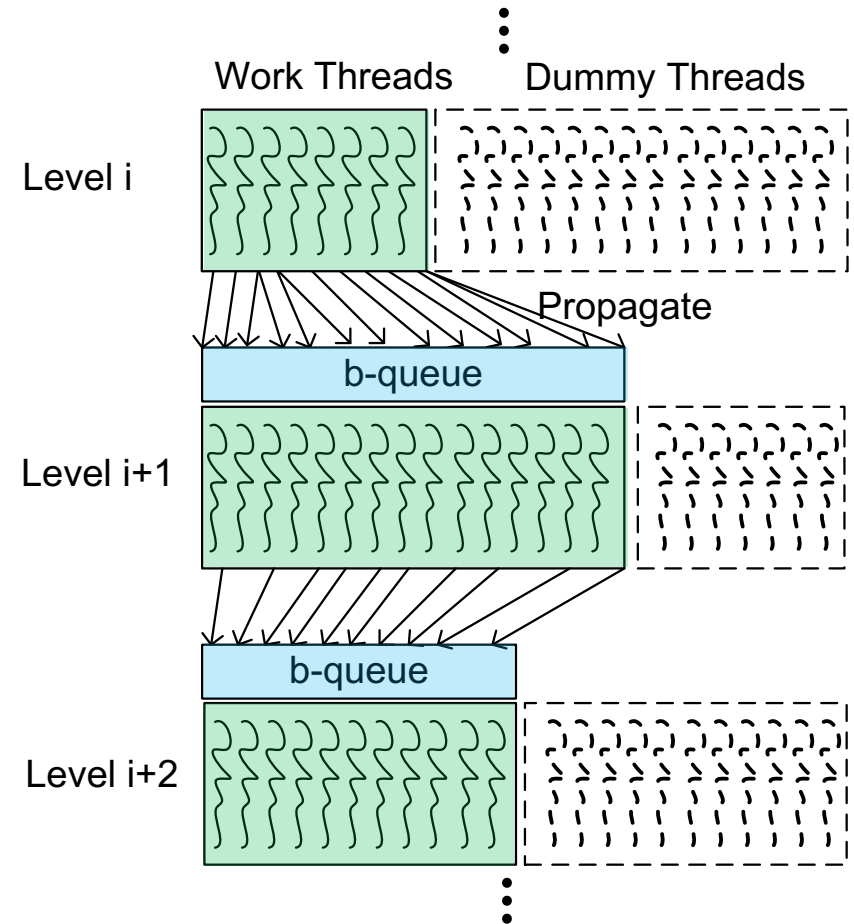
Kernel 2: Kernel re-launch



One-level parallel propagation (i.e., iteration)

Kernel Arrangement (I)

- Kernel 1: **small-sized frontiers**
 - ❑ Only launch one block
 - ❑ Use `__syncthreads()`;
 - ❑ Propagate through multiple levels
 - ❑ Only b-queue
 - No g-queue during propagation
 - Save global memory access
 - Very fast



Kernel Arrangement (II)

- Kernel 2: **big-sized frontiers**
 - Use kernel re-launch to implement synchronization
 - The kernel launch overhead is acceptable considering the time to propagate a huge frontier
- Or, one can use dynamic parallelism to launch new kernels from kernel 1 when the number of nodes in the frontier grows beyond a threshold
 - Dynamic parallelism can also help with load balancing

Hierarchical Kernel Arrangement

- Customize kernels based on the size of frontiers
- Use fast barrier synchronization when the frontier is small



Kernel 1: Intra-block synchronization



Kernel 2: Kernel re-launch

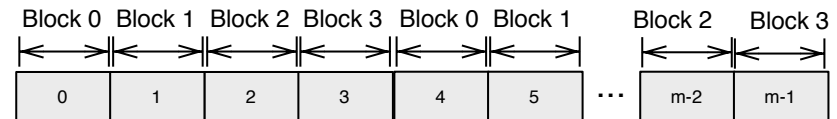
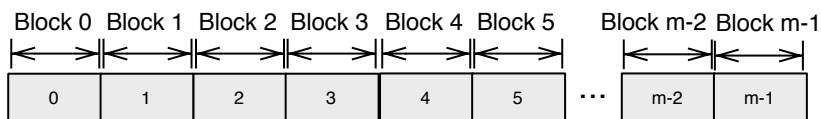
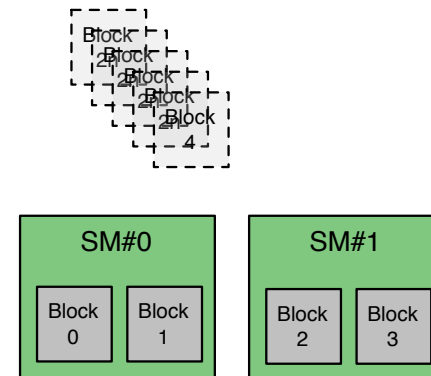
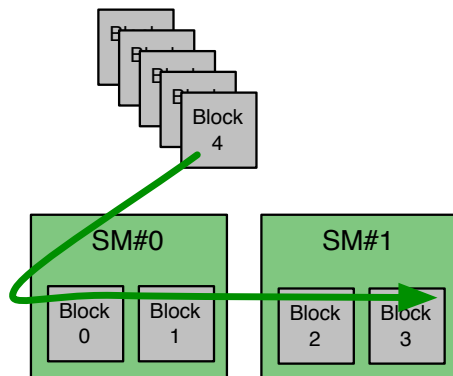


One-level parallel propagation (i.e., iteration)

Persistent Threads and Inter-Block Synchronization

Persistent Thread Blocks

- Combine Kernel 1 and Kernel 2
- We can avoid kernel re-launch
- We need to use persistent thread blocks
 - Kernel 2 launches ($\text{frontier_size} / \text{block_size}$) blocks
 - Persistent blocks: up to $(\text{number_SMs} \times \text{max_blocks_SM})$



Atomic-based Block Synchronization (I)

■ Code (simplified)

```
// GPU kernel
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;

while(frontier_size != 0){

    for(node = gtid; node < frontier_size; node += blockDim.x * gridDim.x){

        // Visit neighbors
        // Enqueue in output queue if needed (global or local queue)

    }

    // Update frontier_size

    // Global synchronization
}
```

Atomic-based Block Synchronization (II)

■ Global synchronization (simplified)

□ At the end of each iteration

```
const int tid = threadIdx.x;
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;
atomicExch(ptr_threads_run, 0);
atomicExch(ptr_threads_end, 0);
int frontier = 0;
...
frontier++;

if(tid == 0){
    atomicAdd(ptr_threads_end, 1); // Thread block finishes iteration
}

if(gtid == 0){
    while(atomicAdd(ptr_threads_end, 0) != gridDim.x){;} // Wait until all blocks finish

    atomicExch(ptr_threads_end, 0); // Reset
    atomicAdd(ptr_threads_run, 1); // Count iteration
}

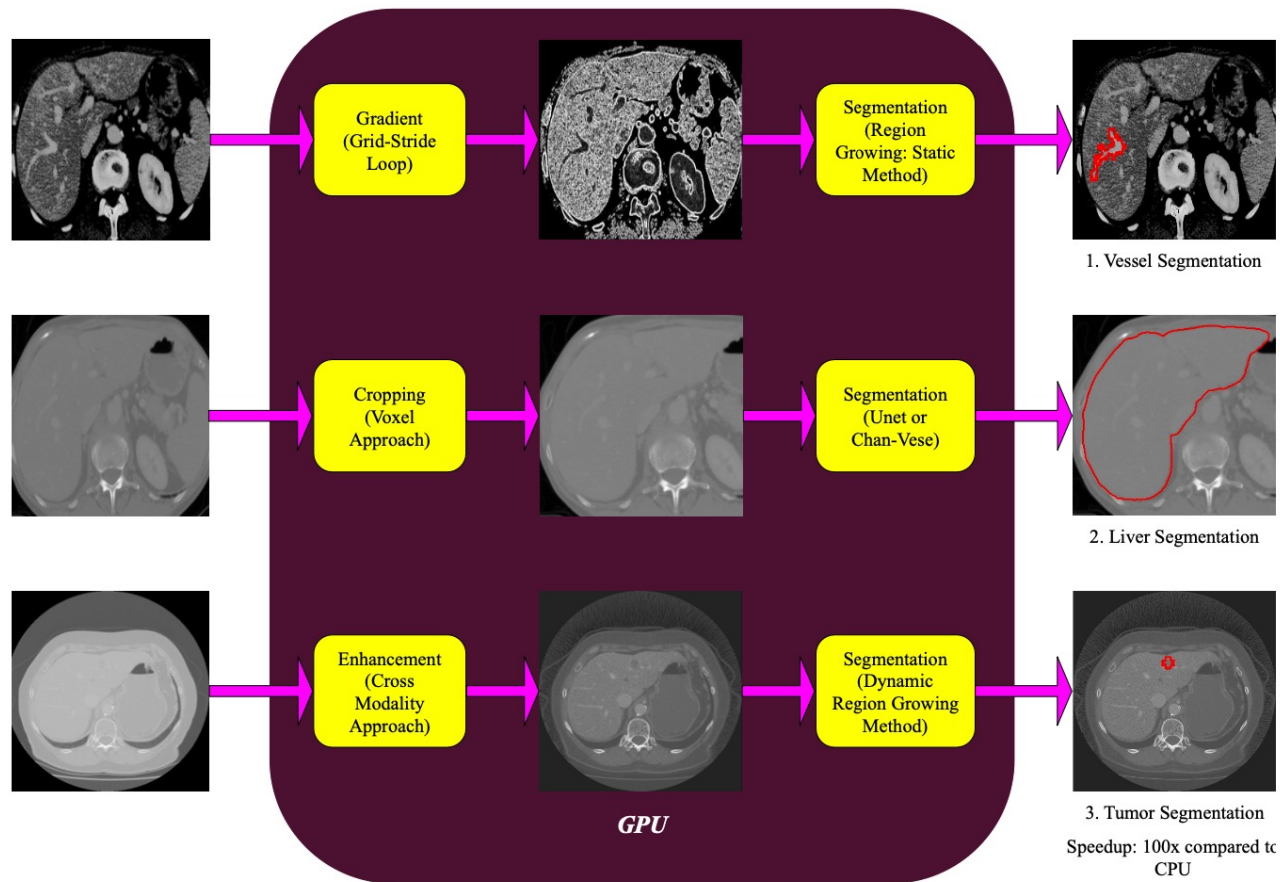
if(tid == 0 && gtid != 0){
    while(atomicAdd(ptr_threads_run, 0) < frontier){;} // Wait until ptr_threads_run is updated
}

__syncthreads(); // Rest of threads wait here

...
```


Segmentation in Medical Image Analysis (I)

- Segmentation is used to obtain the area of an organ, a tumor, etc.



Segmentation in Medical Image Analysis (II)

- Seeded region growing is an algorithm for segmentation
 - Dynamic data extraction as the region grows

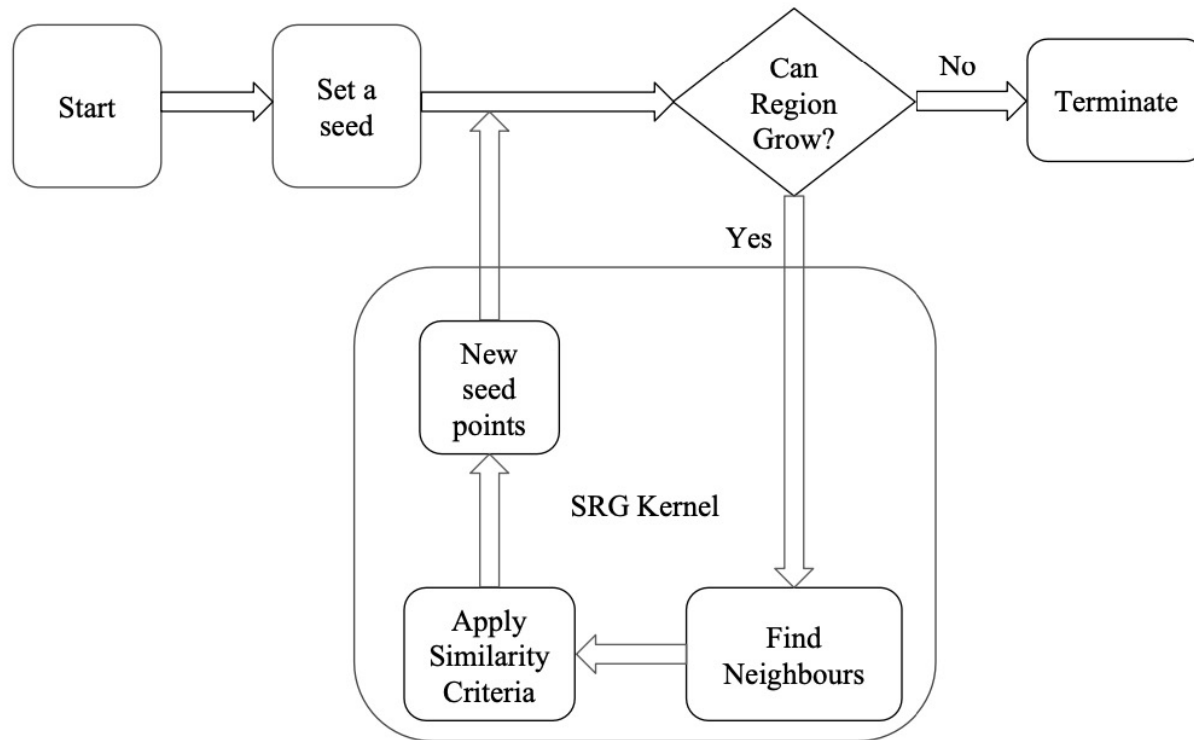
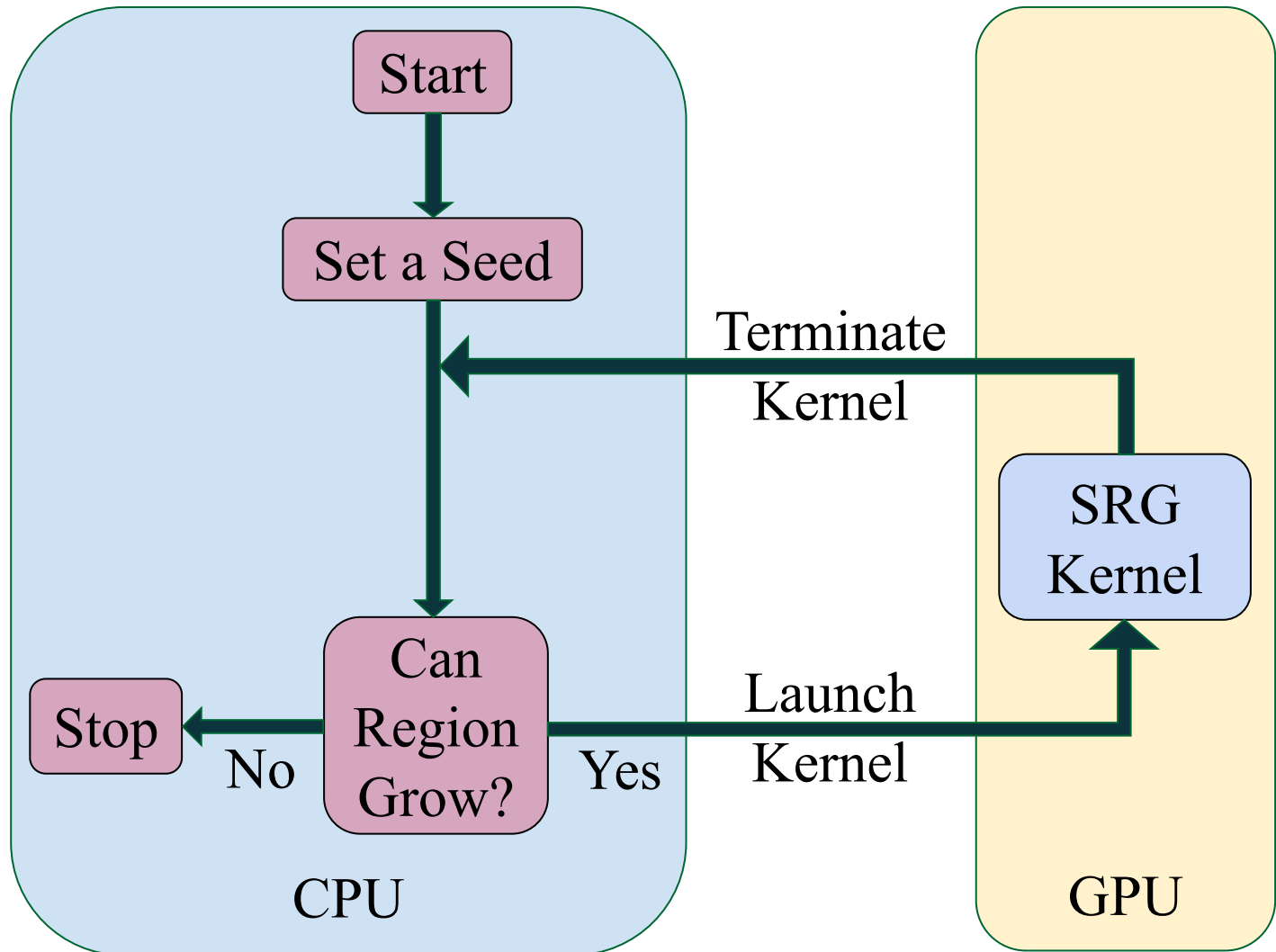
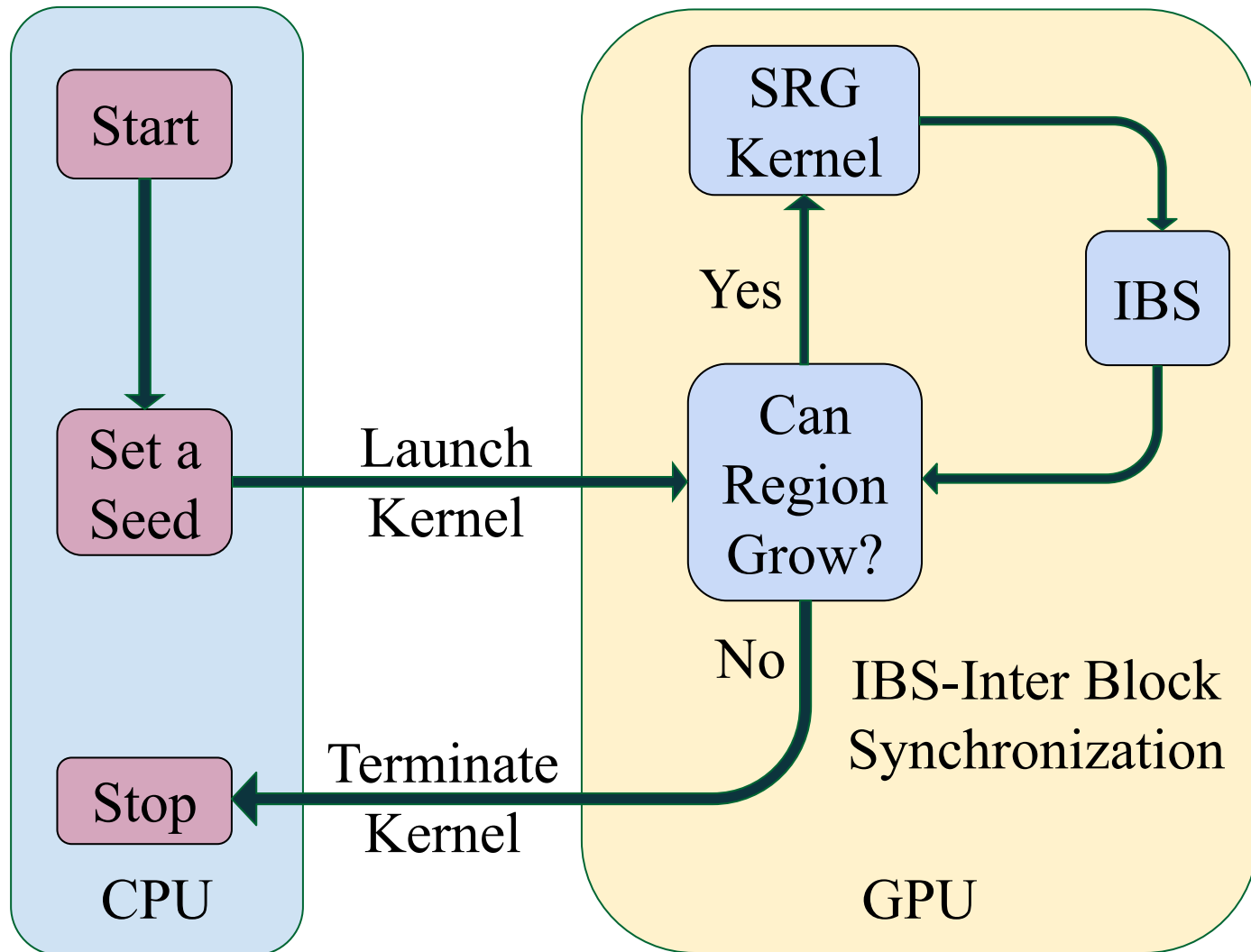


Figure: Seeded Region Growing (SRG)

Region Growing with Kernel Termination and Relaunch



Region Growing with Inter-Block Synchronization



Inter-Block Synchronization for Image Segmentation

Fast parallel vessel segmentation

Nitin Satpute^{a,*}, Rabia Naseem^b, Rafael Palomar^c, Orestis Zachariadis^a, Juan Gómez-Luna^d,
Faouzi Alaya Cheikh^b, Joaquín Olivares^a

^aDepartment of Electronic and Computer Engineering, Universidad de Córdoba, Spain

^bNorwegian Colour and Visual Computing Lab, Norwegian University of Science and Technology, Norway

^cThe Intervention Centre, Oslo University Hospital, Norway

^dDepartment of Computer Science, ETH Zurich, Switzerland

Satpute et al., "Fast Parallel Vessel Segmentation," CMPB 2020. <https://doi.org/10.1016/j.cmpb.2020.105430>

GPU acceleration of liver enhancement for tumor segmentation

Nitin Satpute^{a,*}, Rabia Naseem^b, Egidijus Pelanis^{c,d}, Juan Gómez-Luna^e,
Faouzi Alaya Cheikh^b, Ole Jakob Elle^{c,f}, Joaquín Olivares^a

^aDepartment of Electronic and Computer Engineering, Universidad de Córdoba, Spain

^bNorwegian Colour and Visual Computing Lab, Norwegian University of Science and Technology, Norway

^cThe Intervention Centre, Oslo University Hospital, Oslo, Norway

^dThe Institute of Clinical Medicine, Faculty of Medicine, University of Oslo, Oslo, Norway

^eDepartment of Computer Science, ETH Zurich, Switzerland

^fThe Department of Informatics, The Faculty of Mathematics and Natural Sciences, University of Oslo, Oslo, Norway

Satpute et al., "GPU Acceleration of Liver Enhancement for Tumor Segmentation," CMPB 2020. <https://doi.org/10.1016/j.cmpb.2019.105285>

Accelerating Chan–Vese model with cross-modality guided contrast enhancement for liver segmentation

Nitin Satpute^{a,*}, Juan Gómez-Luna^b, Joaquín Olivares^a

^aDepartment of Electronic and Computer Engineering, Universidad de Córdoba, Spain

^bDepartment of Computer Science, ETH Zurich, Switzerland

Satpute et al., "Accelerating Chan-Vese Model with Cross-modality Guided Contrast Enhancement for Liver Segmentation," CBM 2020.

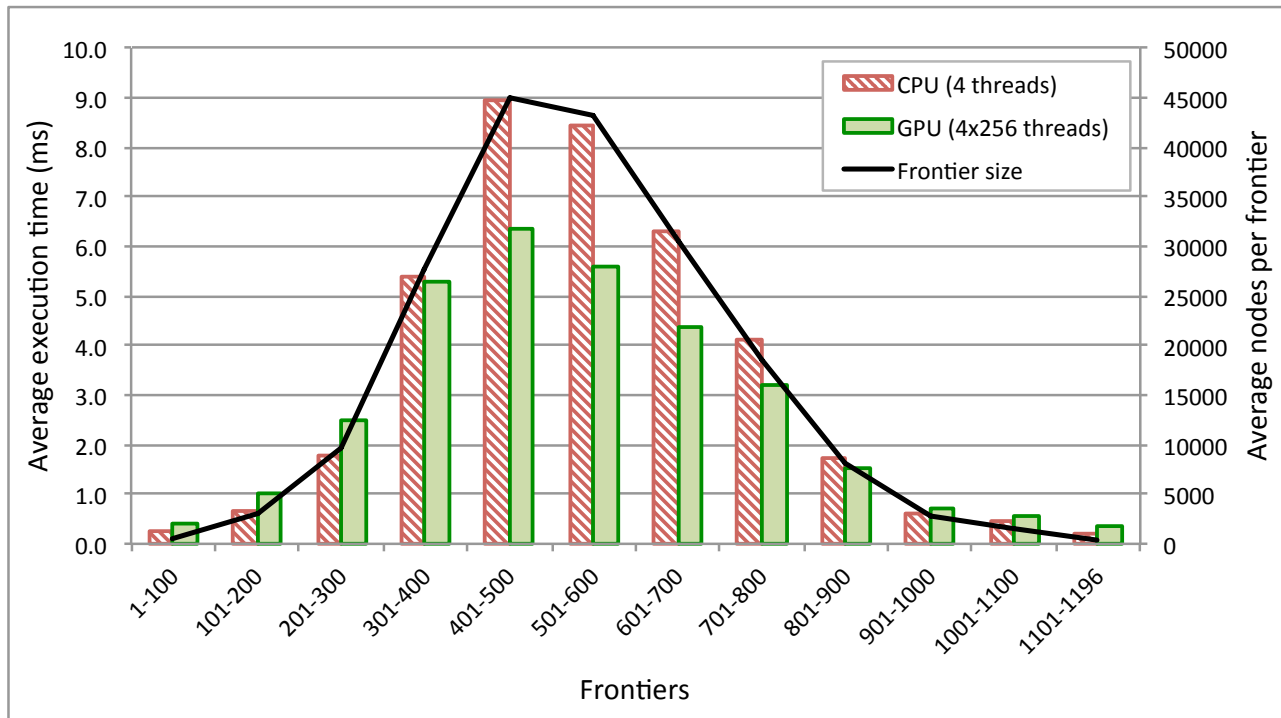
<https://doi.org/10.1016/j.compbimed.2020.103930>

Collaborative Implementation

BFS on CPU or GPU?

■ Motivation

- Small-sized frontiers underutilize GPU resources
 - NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
 - New York City roads



BFS: Collaborative Implementation (I)

- Choose CPU or GPU depending on frontier

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads
    }
    else{

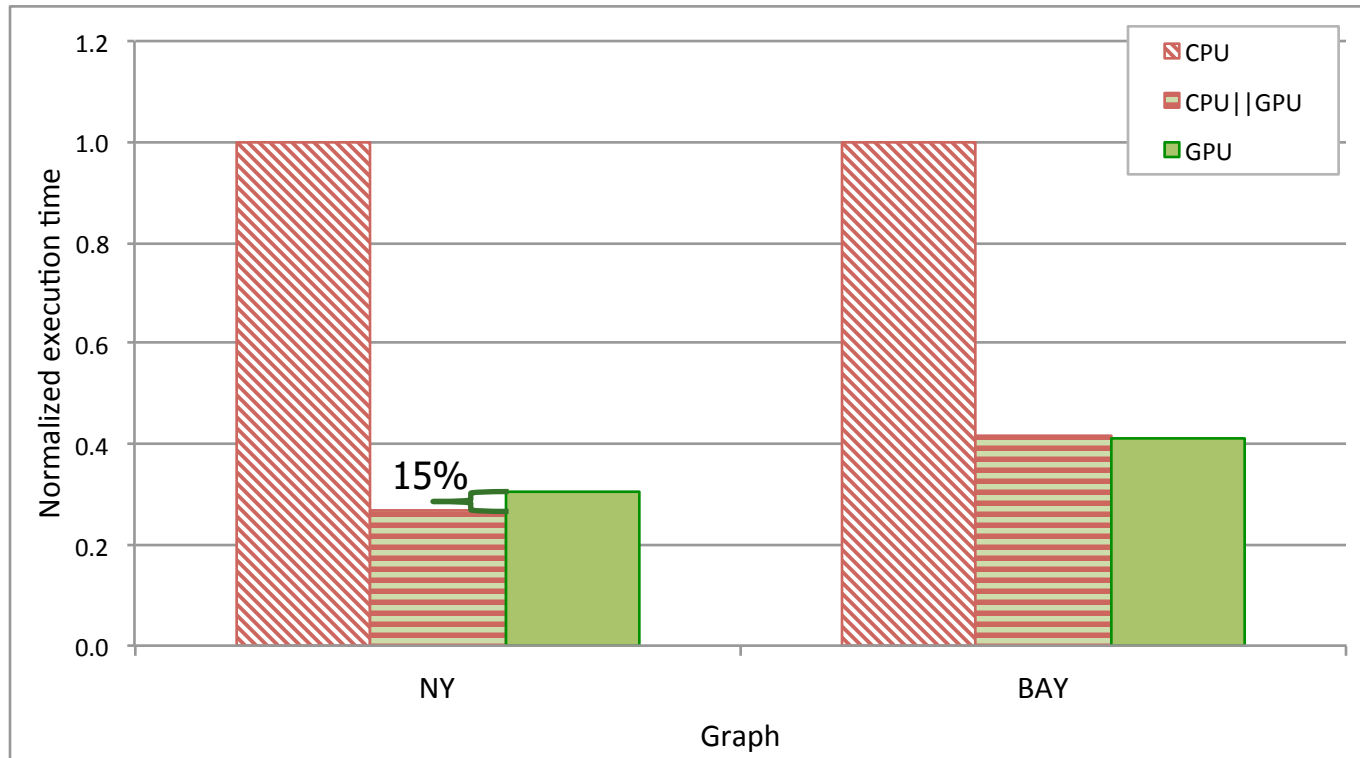
        // Launch GPU kernel
    }
}
```

- CPU threads or GPU kernel keep running while the condition is satisfied

BFS: Collaborative Implementation (II)

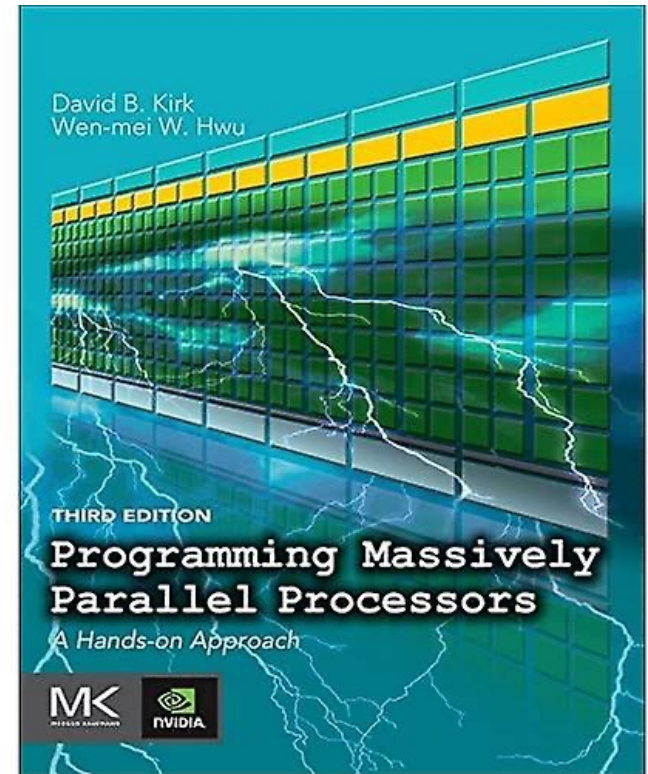
■ Experimental results

- NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)



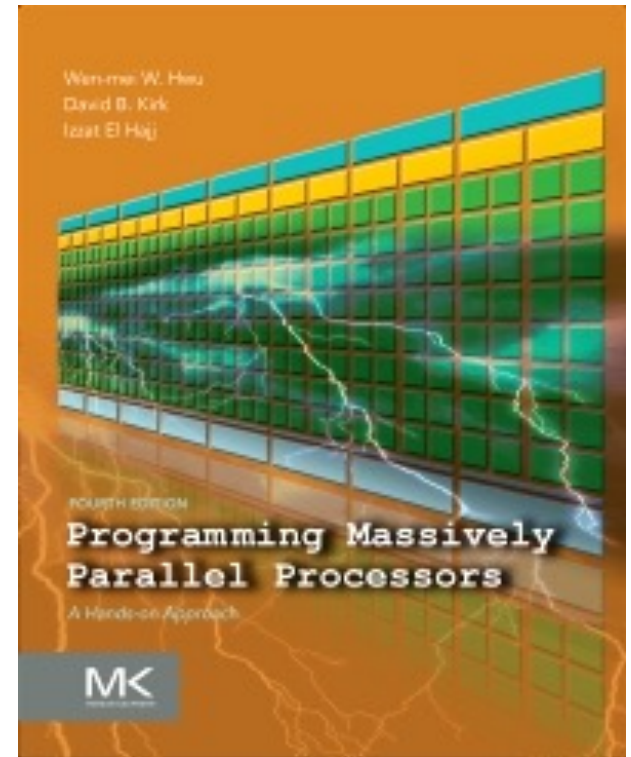
Recommended Readings (I)

- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017
 - Chapter 12 - Parallel patterns: graph search



Recommended Readings (II)

- Hwu and Kirk and El Hajj, “[Programming Massively Parallel Processors](#),” Fourth Edition, 2022
 - Chapter 15 - Graph traversal



P&S Heterogeneous Systems

Parallel Patterns: Graph Search

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

19 December 2022